AFIT/GCS/ENG/91D-20





# EFFECT OF SPATIAL LOCALITY PREFETCHING ON STRUCTURAL LOCALITY

**THESIS**

Dirk D. Schalch, Captain, USAF

AFIT/GCS/ENG/91D-20

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188



| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | December 1991 | Master's Thesis |

**4. TITLE AND SUBTITLE**
EFFECT OF SPATIAL LOCALITY PREFETCHING ON STRUCTURAL LOCALITY

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**
Dirk D. Schalch

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Air Force Institute of Technology, WPAFB OH 45433-6583

**8. PERFORMING ORGANIZATION REPORT NUMBER**
AFIT/GCS/ENG/91D-20

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**
Distribution Unlimited

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

The purpose of this research was to analyze the effect that spatial locality prefetching in cache memory has on the structural locality of program memory referencing behavior. To examine this, a software simulator was built to model a proposed two-level cache memory subsystem. The proposed subsystem was designed to use spatial locality prefetching to exploit the structural locality contained in executing computer workloads. New memory referencing models were developed to incorporate the combined use of structural locality and spatial locality prefetching. From these models, equations were derived to predict the hit rates for both caches. Combined with the state transition probabilities of the memory referencing models, measurements from the trace-driven simulations were used to solve the hit probability equations. This research showed that performance gains through structural locality prefetching are still possible even when spatial locality prefetching is being used in the lower level cache.

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES |
|---|---|
| Cache, Spatial Locality, Structural Locality | 149 |
| | 16. PRICE CODE |

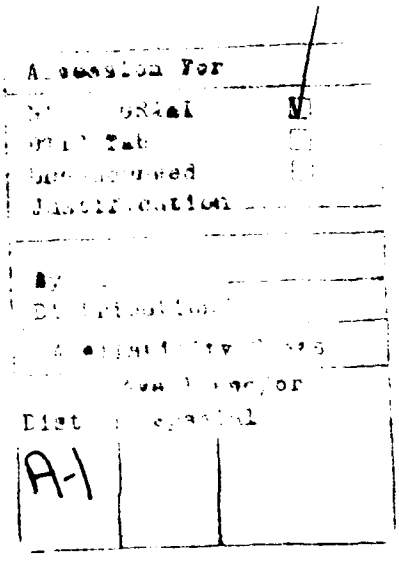
| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

AFIT/GCS/ENG/91D-20

# EFFECT OF SPATIAL LOCALITY PREFETCHING ON STRUCTURAL LOCALITY

**THESIS**

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Systems




**Dirk D. Schalch, B.S.**
Captain, USAF

December 1991

Approved for public release; distribution unlimited

**Acknowledgments**

In accomplishing this thesis, I received invaluable assistance from others. I am greatly indebted to my faculty advisor, Maj William C. Hobart, Jr., for his leadership and support. His belief in me and the importance of my research guided my efforts to completion.

I also would like to thank my thesis committee members, Maj Eric R. Christensen and Maj Kim Kanzaki, for their help. Maj Christensen provided timely assistance with the Ada programming language.

And finally, I wish to thank my wife Maggie whose patience and support made a difficult task much easier to bear.

Dirk D. Schalch

## Table of Contents


| | Page |
|---|---|
| Acknowledgments | ii |
| List of Figures | iv |
| List of Tables | v |
| Abstract | vi |
| I. Introduction | 1-1 |
| II. Literature Review | 2-1 |
| III. Methodology | 3-1 |
| Justification of Methodology Selected | 3-1 |
| Functional Requirements | 3-3 |
| Cache Simulator Preliminary Design | 3-5 |
| Implementation of Cache Simulator in Ada | 3-8 |
| Validation of Cache Simulator | 3-31 |
| Summary | 3-33 |
| IV. Findings | 4-1 |
| Workload Selection | 4-1 |
| Trace-driven Simulations | 4-3 |
| CAM Cache Hit Probability | 4-5 |
| Structural Locality Cache Hit Probability | 4-15 |
| Performance Analysis | 4-22 |
| Summary | 4-23 |
| V. Conclusions and Recommendations | 5-1 |
| Appendix A: Cache Simulator Source Code | A-1 |
| Appendix B: Test Trace Mapping to Testing Requirements | B-1 |
| Appendix C: *Wc* and *Ws* vs. Effective Cache Size Graphs | C-1 |
| Bibliography | BIB-1 |
| Vita | |

## List of Figures


| Figure | | Page |
|---|---|---|
| 2.1 | Two-State Markov Model of Program Behavior . . . . | 2-1 |
| 2.2 | Hobart's Proposed Memory Subsystem . . . . . | 2-6 |
| 3.1 | Cache Simulator Structure Chart . . . . . | 3-7 |
| 3.2 | Linked_List_Package Specification . . . . . | 3-12 |
| 3.3 | CircularQ_Package Specification . . . . . | 3-14 |
| 3.4 | Addr_Record_Package Specification . . . . . | 3-16 |
| 3.5 | Cache_Simulator_Driver Procedure . . . . . | 3-17 |
| 3.6 | Cache Processing Flow . . . . . | 3-21 |
| 3.7 | SLC Prefetch After SLC Miss and CAM Hit . . . . . | 3-23 |
| 3.8 | SLC and CAM Prefetches After Both Caches Missed . | 3-24 |
| 3.9 | Fetch_Address_Package Specification . . . . . | 3-25 |
| 3.10 | Hex_to_Dec_Package Specification . . . . . | 3-26 |
| 3.11 | Determine_Type_Package Specification . . . . . | 3-27 |
| 3.12 | Compute_Miss_Ratios_Package Specification . . . . | 3-28 |
| 3.13 | Compute_Memory_Access_Time_Package Specification . | 3-29 |
| 3.14 | Compute_Cache_Pollution_Package Specification . . | 3-29 |
| 3.15 | Generate_Ref_Frequency_List_Package Specification | 3-30 |
| 3.16 | Simulator Testing Requirements . . . . . | 3-32 |
| 4.1 | Markov Model for CAM Cache Referencing With Prefetching . . . . . | 4-6 |
| 4.2 | Modified Markov Model for CAM Cache Referencing With Prefetching . . . . . | 4-8 |
| 4.3 | Markov Model for SLC Referencing With Prefetching | 4-16 |
| 4.4 | Modified Markov Model for SLC Referencing With Prefetching . . . . . | 4-18 |

## List of Tables


| Table | | Page |
|---|---|---|
| 3.1 | Cache Simulator Requirements Matrix . . . . . | 3-9 |
| 4.1 | Symbolic Workloads Used in Simulations . . . . . | 4-2 |
| 4.2 | SLC and CAM Cache Parameters . . . . . | 4-3 |
| 4.3 | Cache Performance Statistics . . . . . | 4-5 |
| 4.4 | State Transition Probabilities - All References . | 4-10 |
| 4.5 | CAM Cache Hit Probability Comparisons . . . . . | 4-15 |
| 4.6 | SLC Hit Probability Comparisons . . . . . | 4-21 |
| 4.7 | Speedup Due to Spatial and Structural Prefetching | 4-23 |

## Abstract

The purpose of this research was to analyze the effects that spatial prefetching in cache memory have on the structural locality of program memory referencing behavior. To examine this, a software simulator was built to model a proposed two-level cache memory subsystem. The proposed subsystem was designed to use spatial prefetching to exploit the structural locality contained in executing computer workloads.

New memory referencing models were developed to incorporate the combined use of structural locality and spatial locality prefetching. From these models, equations were derived to predict the hits rates for both caches. Combined with the state transition probabilities of the memory referencing models, measurements from the trace-driven simulations were used to solve the hit probability equations.

This research showed that performance gains through structural locality prefetching are still possible even when spatial locality prefetching is being used in the lower level cache.

# Chapter 1

# Introduction

## 1.1 Background

One of the significant factors in improving the performance of a computer system is minimizing the time required to access instructions and data in main memory. Cache memory performs this vital function. Located between the computer processor and main memory, cache memory is small, high-speed memory designed to temporarily store portions of main memory most likely to be referenced by the computer in the near future. Cache memory can typically reduce access time to instructions and data to 10-25 percent of the time to directly access main memory (Smith, 1982:473).

Its extremely fast operating speeds require cache memory to be implemented by special hardware containing high-speed logic circuits (Hayes, 1988:443). As a result, cache memory is very expensive. The challenge for the cache memory designer is to minimize design cost while maximizing cache performance.

One of the key considerations in designing cache memory is understanding the effects that computer workloads can have on cache memory performance (Hobart, 1989:4). Despite its small size, cache memory is able to successfully perform its

functions due to the locality characteristics of workload execution. Three types of program locality are spatial, temporal, and structural (defined in section 1.5). By characterizing the memory referencing behavior of expected computer workloads, one can optimally design a cache memory subsystem which increases computer performance by reducing memory access time.

To take advantage of these referencing localities, cache memory can prefetch in blocks of instructions and data from main memory. Based on current memory referencing, these cache blocks have a high probability of satisfying subsequent memory references (cache hits).

While program locality ensures cache performance, a trade-off exists in how much to prefetch. If the block size is small, reduced bus bandwidth could boost cache performance through decreased transfer time. However, the block size may not be exploiting the locality potential in the workloads. In turn, cache miss ratios could increase. On the other hand, a large block size could improve hit rates by capturing more of the available locality. But this advantage could be hampered by reduced effective cache size resulting from prefetching unneeded references (known as cache pollution).

From this discussion, it becomes apparent that block size prefetch strategies play an important role in determining the effectiveness of a cache memory design.

## 1.2 Statement of Problem

The purpose of my thesis research is to analyze the effects that spatial locality prefetching has on structural locality. This research focuses on the cache pollution which occurs when spatial prefetching is used in a two-level cache hierarchy.

## 1.3 Research Objectives

This research involves the two-level cache memory subsystem proposed by Hobart (Hobart, 1989:96-99). The proposed design (discussed in detail in next chapter) uses two caches to further reduce memory access time.

The cache hierarchy employs spatial prefetching in the secondary cache (closest to main memory). This action attempts to capture any structural locality being exhibited by the executing workload. These referenced structures are then prefetched into a smaller, faster first-level cache located between the secondary cache and the processor.

In addition, Hobart developed four Markov models to represent the referencing behavior of both caches employing prefetching and no prefetching (Hobart, 1989:100-112). From these models, cache hit probability equations were derived. This research analyzes the two Markov models involving prefetching in both caches (discussed in Chapter 4).

The objectives of this research are as follows:

- To design, build, and implement a cache simulator

to represent Hobart's two-level cache hierarchy,

- To use trace-driven simulation to measure the following cache performance statistics resulting from various spatial prefetching strategies: miss ratios, pollution, and effective memory access time.
- To determine how the two Markov models for prefetching can be modified to account for the effects of spatial prefetching,
- To derive cache hit probability equations from these modified Markov models,
- To incorporate the cache pollution measurements into these hit probability equations.

And in so doing, this research

- Provides a documented analysis of the effects that spatial prefetching has on structural locality,
- Provides an analytic model which comprehensively incorporates the effects of spatial prefetching on the two-level cache performance,
- Provides a method to predict cache hit probabilities using measured pollution rates,
- Identifies optimal prefetch block sizes for given cache sizes which could serve as design parameters for a possible hardware implementation of the proposed memory subsystem.

## 1.4 Research Questions

The questions involved in this research are as follows:

- Can the cache simulator be structurally developed in the Ada language and provide acceptable simulation processing speed?
- Does spatial prefetching into the secondary cache effectively capture referenced structures (structural locality) inherent in the workloads? And in the process, how is cache performance affected by any resulting pollution?
- Does structural prefetching into the first-level cache produce an acceptable level of performance? How has this performance been affected by any resulting pollution?
- Can the cache pollution measurements obtained from spatial prefetching strategies be used to predict the hit ratios for both caches?

## 1.5 Definitions

### 1.5.1 Block

A *block* (also referred to as *line*) is a unit of cache memory storage identified by a tag. Block size is always a power of two.

### 1.5.2 Miss

The event that a requested memory address is not

available when referenced in a given cache memory level. If a *miss* occurs in main memory, it is known as a page fault. A *hit* represents the opposite event: requested memory address is available.

### 1.5.3 Miss Ratio

The number of misses occurring in a given cache divided by the total number of memory references to that cache. The *miss ratio* is a cache performance metric. The *hit ratio* represents the opposite metric: number of hits in a cache divided by total number of references to that cache.

### 1.5.1 Pollution

As defined by Smith, *cache pollution* is the portion of prefetched data which is never referenced while residing in the cache (Smith, 1982:482). Cache pollution reduces the cache's effective size.

### 1.5.4 Prefetching

*Prefetching* is the transfer of a block of instructions or data from one level of the memory hierarchy (such as main memory) to a higher level (such as cache memory) prior to being used at that higher level. Unless otherwise noted, prefetch block size will always equal the cache block size.

### 1.5.2 Spatial Locality

Spatial locality is the condition that subsequent memory references will likely occur in locations close to the current reference. Examples of spatial locality are data files or results of a relational database query clustered by an identifying attribute. Both tend to be stored together physically in memory.

### 1.5.3 Structural Locality

Structural locality is the condition that a given set of memory references will likely be referenced in the same order as previously referenced. Termed by Thazhuthaveetil, structural locality is the newest concept to be studied (Thaz, 1986). An example of structural locality is subroutine which may be repeated several times during program execution.

### 1.5.4 Temporal Locality

Temporal locality is the condition that a current memory reference will be likely referenced again in the near future. An example is a program loop which repeats instructions.

## 1.6 Scope of Research

This research involves designing, building, and implementing the cache simulator according to the behavioral description of Hobart's proposed memory subsystem. Once

coded, the simulator is thoroughly tested to prove correctness of design. This validation process ensures that research findings are based on accurate data.

Once the simulator is developed, trace-driven simulations are used to measure the performance in both caches. The traces are comprised of collected memory references obtained from various computer workloads.

The resulting data is used to characterize the effects of spatial prefetching on structural locality. From this analysis, modified cache behavior models are developed to account for these spatial prefetching effects. Cache hit probability equations are derived from these new models. Within these equations, pollution measurements are used to predict the hit rates for the two caches. Using these hit ratios, effective memory access times are calculated for various spatial prefetching parameters.

This research does not involve measuring the effects of prefetching on the access cycle time of each cache memory. To determine the effective memory access time, the cycle times for the memory levels are based on typical values associated with current technology. In addition, this research does not involve developing the hardware circuit design for any components of the proposed memory subsystem. Instead, it employs software simulation to study the different aspects of cache behavior.

## 1.7 Assumptions and Limitations

- The proposed cache memory subsystem operates in a single processor environment.
- Since the goal of this research is to study the effects of prefetching based on symbolic program behavior, system activities such as context switching and interrupt servicing are not included in the workloads. As a continuation of Hobart's research, this research serves as a baseline from which future studies can analyze the effects of prefetching based on total system behavior. The literature review covers some techniques for incorporating context switching in trace-driven simulations.

## 1.8 Summary

This chapter has provided an overview of this thesis effort. The following chapters cover three main areas. Chapter 2 provides an extensive literature review of applicable research. Next, Chapter 3 describes the methodology used to conduct this research. In particular, this chapter covers the design and development of the cache simulator. Chapter 4 provides a detailed description of the research results. Modified cache behavior models incorporating the effects of spatial prefetching are presented. From these models, cache hit probability equations are derived. Calculated results are then compared with actual simulation

measurements. In addition, this chapter investigates cache performance improvements occurring from spatial prefetching. Finally, Chapter 5 provides the research conclusions. Recommendations for future research are presented.

# Chapter 2

# Literature Review

## 2.1 Locality Characteristics of Symbolic Workloads

Hobart analyzed the spatial, temporal, and structural localities of symbolic workloads (Hobart, 1989). Symbolic processing is associated with artificial intelligence applications.

Using trace-driven simulation, Hobart methodically characterized the locality aspects of symbolic workloads by examining the low-level memory referencing behavior. To examine the "temporal distances" of memory references, Hobart developed a two-state Markov model as shown in Figure 2.1 (Hobart, 1989:40-42). The various state transition probabilities are discussed in Chapter 4.

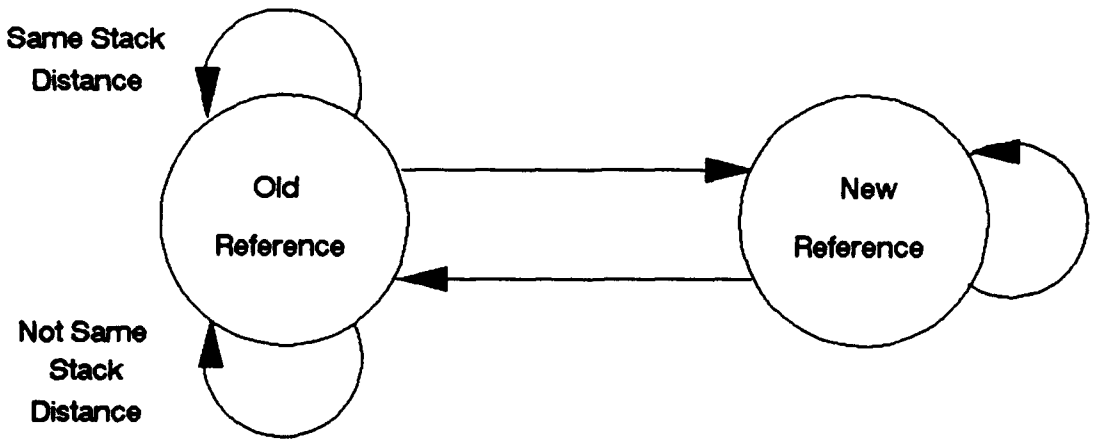



Figure 2.1: Two-State Markov Model of Program Behavior

Previously unreferenced addresses are described by the "*new reference*" state. While previously referenced addresses are depicted by the "*old reference*" state. Transitions between the two states occur as memory referencing shifted from old locations to new locations (vice versa). Within the *old reference* state, consecutive references which occur in the same order in which they were previously referenced are represented as "*same-stack-distance (SSD)*" transitions. The notion of a "*stack*" is used to describe the ordered contents of the cache. Within the cache, stack distance represents the spatial distance from the last address (top of the stack) to another address. Given a previously referenced address, an *SSD* reference takes place when the next reference is located the same spatial distance from the top of the stack as when it was last referenced. In turn, consecutive *SSD* transitions show a rereferencing of cache addresses in the same order as before. Conversely, consecutive old references with different stack distances (not in the same order) are "*not-same-stack-distance (NSSD)*" transitions. In total, Hobart identified five possible state transitions: *New-New*, *New-Old*, *Old-New*, *Old-SSD*, and *Old-NSSD*.

Employing this model, Hobart implemented a systematic method for extensive analysis of program locality. New metrics were developed to measure referencing behavior.

Studying the spatial characteristics of reference strings, Hobart discovered an unique aspect of memory refer-

encing behavior. When an executing workload is exhibiting spatial locality, subsequent references took place within a physical address distance of 32 words from the previous references (Hobart, 1989:51-53). He labeled this characteristic the "spatial locality window." This narrow spatial window was found to exist in all types of workloads both symbolic and non-symbolic (conventional).

From this, a new spatial locality metric was developed called the "spatial window probability (*Psw*)": the probability that given a current address, the subsequent reference is within 32 words. Hobart observed that the *Psw* of symbolic workloads was almost 50 percent greater than of conventional workloads. This was attributed to the higher percentages of instruction fetches inherent to symbolic processing.

As a result of the higher *Psw*, Hobart suggested that spatial prefetching may prove more effective for symbolic workloads. In addition, the narrow spatial window allowed smaller prefetch block sizes which reduces the potential for cache pollution.

To analyze temporal locality characteristics, three metrics were developed to measure the cumulative temporal distances of program referencing (Hobart, 1989:55-69). The *LRU90*, *LRU95*, and *LRU99* metrics represented "simulated, fully-associative least recently used (LRU) stack depths" needed to capture 90, 95, and 99 percent of a workload's old references, respectively.

Hobart found that the temporal distances of symbolic workloads were significantly less than those of conventional workloads. The conventional *LRU99* was five times greater. This behavior was attributed to program referencing characteristics. Symbolic workloads tend to access the first few elements of a list. In contrast, conventional workloads tend to access their entire structures evenly. In addition, symbolic workloads only reference about one third the number of distinct data addresses.

From this, Hobart suggested a trade-off between cache design options tailored toward symbolic workloads. Based on the temporal analysis, a 99 percent hit rate on old references can be attained with a cache one fifth the size that would be required for conventional workloads.

To analyze structural locality characteristics, Hobart developed the *Pssd* metric: the probability that given a previously referenced address, the subsequent reference will be to an address with the "*same stack distance*" to the previous reference (Hobart, 1989:69-71).

Hobart found the *Pssd* depicted one of the most unique aspects of symbolic memory referencing behavior. Over one half of symbolic references (Pssd = 0.550) can be classified as the referencing of ordered structures. In contrast, the percentage for conventional workloads is only slightly more than one fourth.

As a result of his findings, Hobart proposed a memory

subsystem design to exploit the structural locality of symbolic workloads (Hobart, 1989:96-99). The design involves a two-level cache hierarchy comprised of a small structural locality cache (SLC) close to the processor and a larger content-addressable memory (CAM) cache. The proposed design is shown in Figure 2.2.

The main function of the CAM is to capture the structural locality being exhibited by the workload. It can accomplish this goal by using a first-in-first-out (FIFO) circular buffer replacement algorithm. This algorithm allows the CAM to store blocks containing requested addresses in the order received from main memory. As a result of the maintained order, structural locality will remain intact after being fetched into the CAM.

The SLC is then able to exploit this structural locality by prefetching requested blocks from the CAM. Unlike the CAM, the SLC does not require reordering. This reason combined with the small size of the SLC allow an LRU replacement algorithm to be employed.

Both caches are comprised of content addressable memory. This type of memory is fully associative and, in turn, allows addresses to be simultaneously searched. The result is improved access times due to reduced latency.

As discussed in Chapter 1, this proposed memory subsystem will used in this research. One of Hobart's main concerns was cache pollution in the CAM resulting from spatial prefetching.

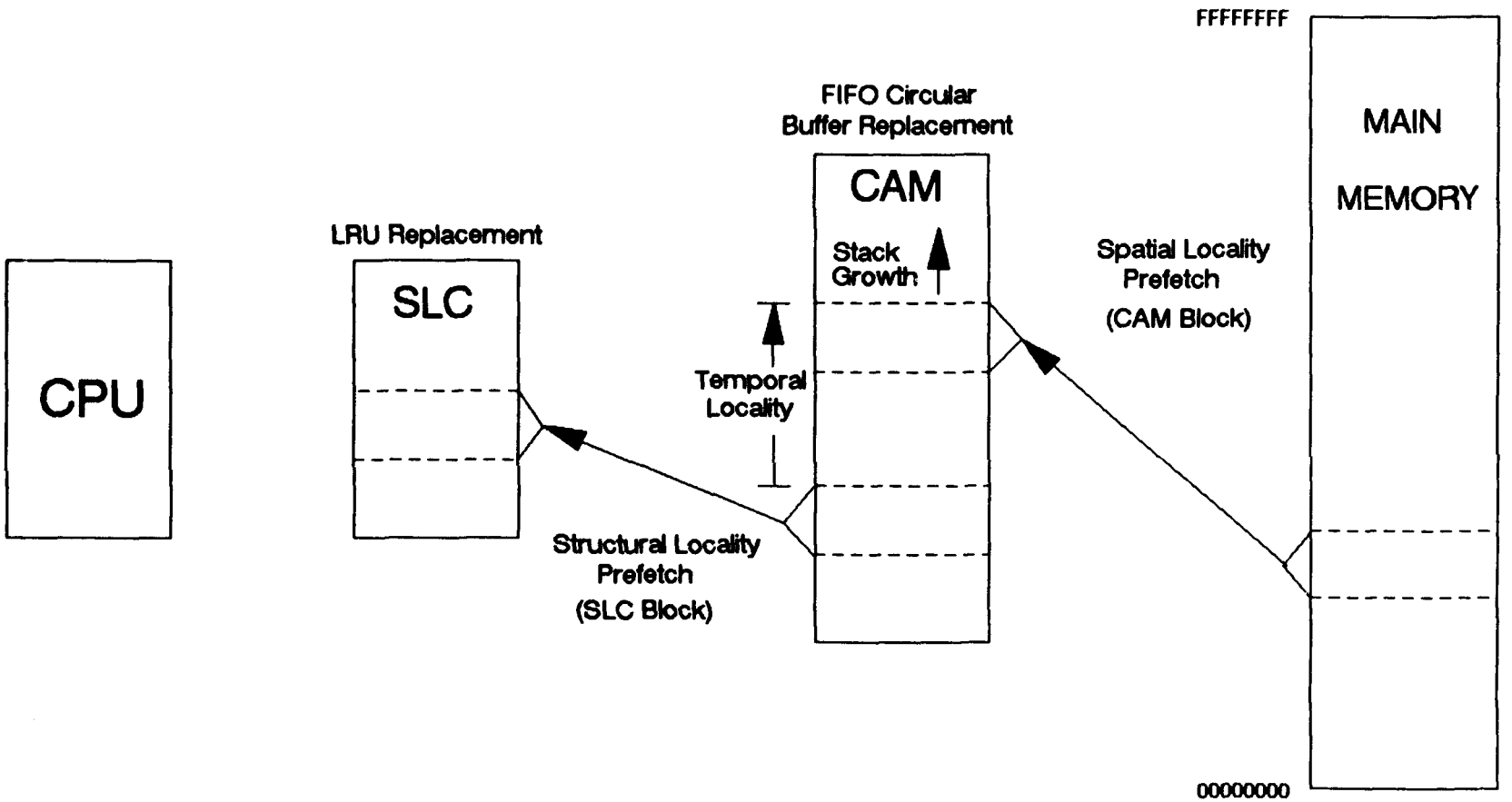



Figure 2.2: Hobart's Proposed Memory Subsystem

How would reduced structural locality resulting from pollution effect the performance of the SLC and CAM caches? This research will focus on this concern.

## 2.2 Trace-Driven Simulation

Trace-driven simulation involves the use of collected sequences of virtual addresses (traces) to drive a simulation model of the cache memory system (Smith, 1982:479-480). By changing parameters within the simulator, the effects of cache design choices (cache size, block size, replacement, etc.) can be studied.

Smith identified two major advantages of trace-driven simulation (Smith, 1987:1065). It allows the analysis of cache memory performance based on actual computer workload behavior. Mathematical models and random number generators have fallen short in representing true program characteristics. The other advantage is flexible and feasible cache design assessment. Hardware prototypes require extensive development time with minimum design variance. In contrast, cache simulators can be developed quicker with maximum design parameter ranges.

Smith pointed out two major limitations of trace-driven simulation (Smith, 1987:1065). He found that actual miss ratios from executing workloads in a real system environment are almost always exceed those obtained through simulations. This difference can be attributed to several reasons. Due to

their relatively small sizes, traces may not provide representative samples of the computer workloads. Operating system activity, such as context switching and interrupt servicing, may not be contained in the trace. In addition, input/output handling may not be included.

The other limitation results in reduced analysis of larger caches. In order to incorporate process switching, Smith employed cache "flushing" to simulate the changing of working sets. However, this approach combined with the limited size of traces prevented larger caches (beyond 32K bytes) from being filled. The result was a lower limit to large cache performance.

To improve the trace collection process, Agarwal, Sites, and Horowitz developed a new technique involving the modification of microcode to capture memory references as they occur (Agarwal and others, 1986:119-127). Using the VAX 8200, they implemented microcode changes which recorded all virtual address references to include process switching and system calls. The method was dependent on available microcode memory. Microcode was modified everywhere a memory reference could be generated. As a result, saturated memory prevented all required microcode changes.

Hobart overcame this memory problem. Using a microcode modification technique similar to Agarwal, Sites, and Horowitz's, Hobart eliminated the need to embed microcode changes at every memory referencing location (Hobart, 1989:31-

32). Instead, the page map table was changed to automatically invoke the "page fault abort handler" on every virtual address reference. In turn, the handler was modified to collect the virtual address traces.

To improve trace data as a true representation of program behavior, Iyer, Laha, and Patel developed a sampling technique to estimate the distribution of cache miss ratios due to task switching (Iyer and others, 1988:1325-1330). Similar to Smith's approach, their technique uses an emptying of the cache to represent a context switch. However, while Smith simulated task switching by purging the cache at consecutive intervals, Laha, Patel, and Iyer sampled a trace at points where task switches occurred.

Their methodology involves the following steps. First, they chose a sample size which represented the length of the process interval. Next, based on trace size, the sampling frequency was established to obtain a target number of samples. Once these parameters were set, the trace samples were collected so that the start of each sample mapped to a context switch. When running the simulation, the cache memory would be purged at the start of each sample to coincide with the task switch.

The result is a trace-driven simulation which produces a more accurate distribution of the cache miss ratio. Typical samples sizes were 5000, 10000, and 20000 address references. Laha, Patel, and Iyer used 35 samples from each trace to

attain an acceptable level of confidence.

## 2.3 Impact of Prefetching on Cache Performance

The locality of memory referencing in executing workloads provides the opportunity to predict which portions of the address space will most likely be referenced next. Cache memory takes advantage of this locality by prefetching instructions and data ahead of their actual usage.

As Smith pointed out, the result is substantial improvement of system performance (Smith, 1978:7-21). The other choice is to fetch on demand. Smith explained that demand fetches incur high penalties in CPU overhead and idle time. In a single process environment, the CPU must wait while transfers between cache and main memory are accomplished. Even with multiprogramming, a process may not always be ready to execute while memory requests are being satisfied. In addition, having to schedule and start each separate demand fetch can lead to increased overhead per transfer.

In determining the effectiveness of prefetching, Smith compared the reduction of the miss ratio for a given block size to the corresponding increase in transfer ratio. Transfer ratio was comprised of the miss ratio and prefetch ratio: "number of prefetch data transfers to total number of memory references." Smith concluded that prefetching can reduce the cache miss ratio at a cost level less than the percentage increase in transfer ratio. He found that a block

fetch size of 32 bytes was generally effective for cache memory sizes up to 64K bytes. A 64-byte block size produced comparable (even better) miss ratios but at the expense of increased transfer costs. In this research, a CAM block size of 32 bytes is used with a 32K byte CAM for one set of simulations.

In their research of cache performance in a Unix environment, Alexander, Keshlear, Cooper, and Briggs also looked at prefetching effects on bus traffic (Alexander and others, 1986:41-70). Similar to Smith, 32 bytes proved to be an effective prefetch block size. They found that block sizes ranging from 8 to 32 bytes resulted in the largest reductions in bus traffic.

Smith identified several architectural factors which can affect block size choice (Smith, 1987: 1064). "*Memory interference*" and "*memory busy time*" can result from larger block sizes. In multiprocessor systems, the longer line can tie up the memory and bus and, in turn, adversely impact other processors. In addition, "*I/O overruns*" may occur. Memory interference may cause I/O operations to be aborted and reinitiated. Another factor involves address tag storage. If a block size is small, a substantial amount of the cache storage is required for the address tags. The result is the effective size of the cache is decreased. One more factor concerns *copy-back* caches. A larger block size can increase memory traffic for each copy back. However, Smith suggests

the additional traffic can be countered by the lower miss rates attained from larger lines.

To improve multiprocessor system performance, Johnson also suggests the use of prefetching to reduce contention for shared memory resources (Johnson, 1989:137-141). He explains an approach called "tagged working set prefetching." Each prefetched block carries a tag to uniquely identify its working set. The prefetched blocks are then "broadcast" to all processors. Using the tags, other cache controllers can load any required broadcast blocks. The result is reduced memory traffic by accomplishing several prefetches with single accesses.

Smith attributed the effect on miss ratio as the main influence that block size choice has on cache performance (Smith, 1987:1064-1074). Using extensive trace-driven simulation, Smith observed that larger block sizes generally reduced cache miss ratios. Longer lines tend to exploit the memory referencing localities of executing workloads. The upper limit to this effect occurred when the block size approached the cache size. After this point, the miss ratios increased. Smith explained that the increase was due to less captured program locality resulting from decreased number of cache blocks. For cache sizes of 32K and 64K bytes, prefetch block sizes of 16-64 bytes continued to produce the lowest miss ratios. In addition to the simulations involving a 32 byte CAM block size, another set of simulations used in this

research employs a 16 byte block size for a 32K byte CAM cache.

Using trace-driven simulations involving similar cache sizes, Przybylski also produced optimal prefetch block sizes of 32-64 bytes (Przybylski, 1990:160-169). These sizes resulted in the minimum effective memory access times.

The advantages of prefetching extend to all levels of the memory hierarchy. Excessive dependence on disk I/O operations can significantly decrease system performance. Smith identified database systems as excellent candidates for data caching (Smith, 1978:223-246). The high degree of sequential access inherent in database systems allow subsequent data references to be predicted. In turn, data blocks or segments can be prefetched into main memory to satisfy future requests. The resulting reduction in disk I/O substantially improves database system performance.

## 2.4 Multi-Level Cache Hierarchies

With the speeds of new processors continuing to increase, traditional single-level caches will not be able to exploit the extremely fast CPU cycle times. As Smith pointed out, expanding the size of a single cache produces two problems (Smith, 1982:517). One is large physical size and circuit complexity increases memory access time. The other problem is adding more cost to an already expensive hardware component.

Hennessy, Horowitz, and Przybylski observed that once the

cache size reaches 64KB, there exists little margin for performance improvement (Hennessy and others, 1988:290-298). As they note, "chip to chip communication" and control circuitry continue to contribute a major portion of memory latency. Physical cache size and material properties limit transfer rates. Consequently, as CPU cycle time grows faster, the single-level cache is hard pressed to maximize CPU performance.

Multi-level caches provide a solution. Employing smaller, faster cache design technology, cache hierarchies can locate a first-level cache close to the processor to lower memory latency and transfer times. As Hennessy, Horowitz, and Przybylski pointed out, the addition of a second-level (*C2*) cache substantially decreases the miss penalty of the first-level (*C1*) cache (Hennessy and others, 1989:114-120). The result is lower effective memory access time leading to increased CPU performance.

In their two-level cache simulation study, Levy and Short show that the addition of a secondary cache can boost system performance (Levy and Short, 1988:81-87). Employing trace-driven simulations, they compared the execution cycle times of two-level caches to those of a single-level cache. Results revealed that given an *C2* to *C1* size ratio of at least 8:1, system performance was significantly improved from the addition of the *C2* cache. For example, given a 15-cycle main memory access time, combining a 4-cycle, 256KB *C2* cache with

a 1-cycle, 8KB *C1* cache produced a performance increase of 18 percent. In this research, an 8:1 ratio of CAM to SLC size is used in the simulations.

Using trace-driven simulations of two-level caches, Hennessy, Horowitz, and Przybylski showed that a *C1* cache largely decreased the number of memory references to the *C2* cache (Hennessy and others, 1989:114-120). They noted that the smaller amount of *C2* references reduces the impact of *C2* cycle time. As a result, optimal secondary cache sizes can exceed the sizes of single-level caches. Interesting similarities were found between the "*global*" miss ratios (number of misses divided by total CPU references) of secondary caches and the miss ratios of corresponding single-level caches. They determined that if the *C2* cache is at least eight times the size of the *C1* cache, then the global miss ratio of *C2* is basically equivalent to the miss ratio of the comparably-sized single cache.

From trace-driven simulations of two-level caches, Bakka, Bugge, and Kristiansen examined the miss ratios of secondary caches (Bakka and others, 1990:250-259). Their simulations involved *C2* cache sizes of 1-8 MB with the smallest size being eight times the size of the *C1* cache. They found that *C2* block sizes of 128 and 256 bytes produced the lowest miss ratios in the secondary caches.

Due to the high hit ratios of the first-level cache, improving the access time of this cache is an important design

goal. Baer, Levy, and Wang proposed a two-level cache hierarchy designed to minimize the access time of the *C1* cache (Baer and others, 1989:140-148). They suggested that the *C1* cache can be optimally accessed by virtual addresses. In turn, the *C1* access time is reduced since no address translation is required. Address translation and handling of "synonyms" (copies of data under different virtual addresses) is accomplished in the *C2* cache. As discussed earlier, decreased referencing of the *C2* cache lessens the impact of increased cycle time overhead. From their simulation results, they concluded that if the address translation penalty in the *C1* cache is at least six percent, then switching to the virtual *C1* design would reduce effective memory access time.

## 2.5 Summary

Several cache performance characteristics identified in this chapter have direct applicability to this research. Increasing the cache size beyond 64K bytes results in little improvement to the hit ratio. For cache sizes of 32K and 64K bytes, prefetch block sizes of 16-64 bytes produce the optimal hit ratios. In two-level cache hierarchies, the size ratio of the secondary cache to the primary cache should be at least 8 to 1. These performance characteristics are used to establish the cache size and block size parameters for the trace-driven simulations.

# Chapter 3

# Methodology

This chapter describes the methodology employed to design and build the two-level cache simulator used for this research. It starts with a justification of the methodology selected. An overview of the experimental setup is provided. Next, the cache simulator design and implementation is covered in detail. The chapter ends with a description of the test procedures used to validate and verify the correctness of the simulator functions.

## 3.1 Justification of Methodology Selected

As covered in the background, trace-driven simulation is the technique to be used to evaluate the proposed memory subsystem design. Hobart developed a simulator for the two-level SLC and CAM cache hierarchy. This simulator was written in Lisp and was built to run on the TI Explorer II computer.

For this research, the decision was made to design and develop another simulator for the SLC and CAM cache memory subsystem. This simulator is written using the Ada programming language.

The rationale for this decision is based on several justifications. One main reason is to take advantage of the

powerful, high-speed Sun SPARC microprocessor workstations. The idea is very simple: to minimize the execution time of the trace-driven simulations. The speed of the Sun SPARC microprocessor is orders of magnitude (over 20 to 1) faster than the TI Explorer II. Importantly, the Sun SPARC workstation contains an extensive Ada software development/ execution environment: Verdix Ada Development System (VADS - Version 6.0)

Another justification is based on the choice of the Ada programming language. From a development standpoint, Ada offers several advantages. A main strength of Ada lies in its handling of abstract data types. Using a "packaging" concept, Ada provides the ability to "encapsulate" abstract data types along with the operations which manipulate their states (Feldman, 1985:4-7). This Ada feature allows the cache simulator to be developed using software packages which can be integrated into a reliable, structured design. In addition, this advantage facilitates modification and expansion of the simulator to incorporate growing research requirements. New aspects of cache behavior can be analyzed by modifying existing or developing additional packages.

Another advantage is reusability. The use of Ada constructs to achieve a highly-structured design promotes an understanding of the software system. In turn, future research efforts can benefit from expanded versions of the cache simulator.

## 3.2 Functional Requirements

The following requirements served as the functional baseline for the SLC and CAM cache simulator.

### 3.2.1 Workload Traces

Several sets of virtual address traces were available for use in this research. Sixteen workload traces were collected by Agarwal, Sites, and Horowitz from the VAX 8200 (Agarwal and others, 1986). Hobart provided 15 address traces collected on the TI Explorer II (10 traces) and the IBM System/360 Model 91 (5 traces).

The cache simulator must accept as input the memory references contained in these traces. The memory references in the Agarwal, Sites, and Horowitz traces are in hexadecimal format. In addition, each memory reference has an integer number identifier. This identifier indicates whether the address is a data read, data write, or instruction fetch. The memory references in the Explorer II/IBM 360 traces are in integer format. These traces are available in separate versions: data read only, data write only, instruction fetch only, data reference only, and all references. As a result, they do not contain reference identifiers.

### 3.2.2 Cache Design Parameters

The cache simulator must allow modification of the following parameters for varied simulations: SLC size, SLC

prefetch block size, CAM size, and CAM prefetch block size. The SLC shall employ a LRU replacement algorithm. The CAM shall employ a FIFO circular buffer replacement algorithm.

### 3.2.3 SLC and CAM Miss Ratios

For each trace-driven simulation, the cache simulator must calculate the miss ratios for the SLC and CAM caches.

### 3.2.4 SLC and CAM Memory Access Time

For each trace-driven simulation, the cache simulator must calculate the effective memory access time. If we assume that a cache block transfer can be overlapped with the CPU execution, then the effective memory access time $t_a$ is:

$$t_a = t_{SLC}p_{SLC} + t_{CAM}(1 - p_{SLC})\,p_{CAM} + t_{MEM}(1 - p_{CAM})\,(1-P_{SLC})$$

where

$t_{SLC}$ = *memory access time of the SLC*

$p_{SLC}$ = *hit rate of the SLC*

$t_{CAM}$ = *memory access time of the CAM*

$p_{CAM}$ = *hit rate of the CAM given a SLC miss*

$t_{MEM}$ = *memory access time of the main memory*

### 3.2.5 SLC and CAM Cache Pollution

For each trace-driven simulation, the cache simulator must calculate the SLC and CAM cache pollution percentages.

### 3.2.6 Reference Frequency

For each trace-driven simulation, the cache simulator must individually track how long it takes (number of references) before each prefetched address is first referenced. Cache pollution references will not be included. For each given number of references, the cache simulator must determine how many prefetched addresses required the same number of references. For example, 1000 prefetched addresses in the SLC took 50 memory references before they were referenced for the first time.

## 3.3 Cache Simulator Preliminary Design

The purpose of this stage was to develop a high-level architecture depicting how the simulator can be structured to meet all functional requirements. In turn, this architecture served as guide to producing an Ada software design solution.

The design approach used for this cache simulator is based on functional decomposition. To achieve a structured design, the simulator is comprised of an integrated set of program modules. Each module performs functional requirements involving the processing of input(s) to produce required output(s).

A structure chart is constructed to provide a pictorial representation of how the cache simulator program modules would work together. The structure chart offers a way to develop the simulator architecture without requiring knowledge of the internal workings of each module. The syntax involves using boxes to represent the program modules. Communication between the modules is represented by labeled arrows.

Figure 3.1 shows the structure chart for the cache simulator. As depicted, the program modules are grouped into three major areas: input, processing, and output. The top box represents the cache simulator driver module which directs the activity between the three areas.

The input area is comprised of three modules. The *Fetch Trace Address* module extracts one memory reference address at a time from the trace file. Prior to being sent to the driver, the memory reference is converted from hexadecimal to integer format by the *Convert Hex Address to Integer* module. The *Determine Reference Type* module determines whether the memory reference is a data read, data write, or instruction fetch.

The cache processing area performs the SLC and CAM cache functions. It is comprised of the following three modules: *Process Data Read Address*, *Process Data Write Address*, and *Process Instruction Fetch Address*. Depending on the memory reference type, the driver invokes the appropriate module to service the trace address. Since these modules execute the

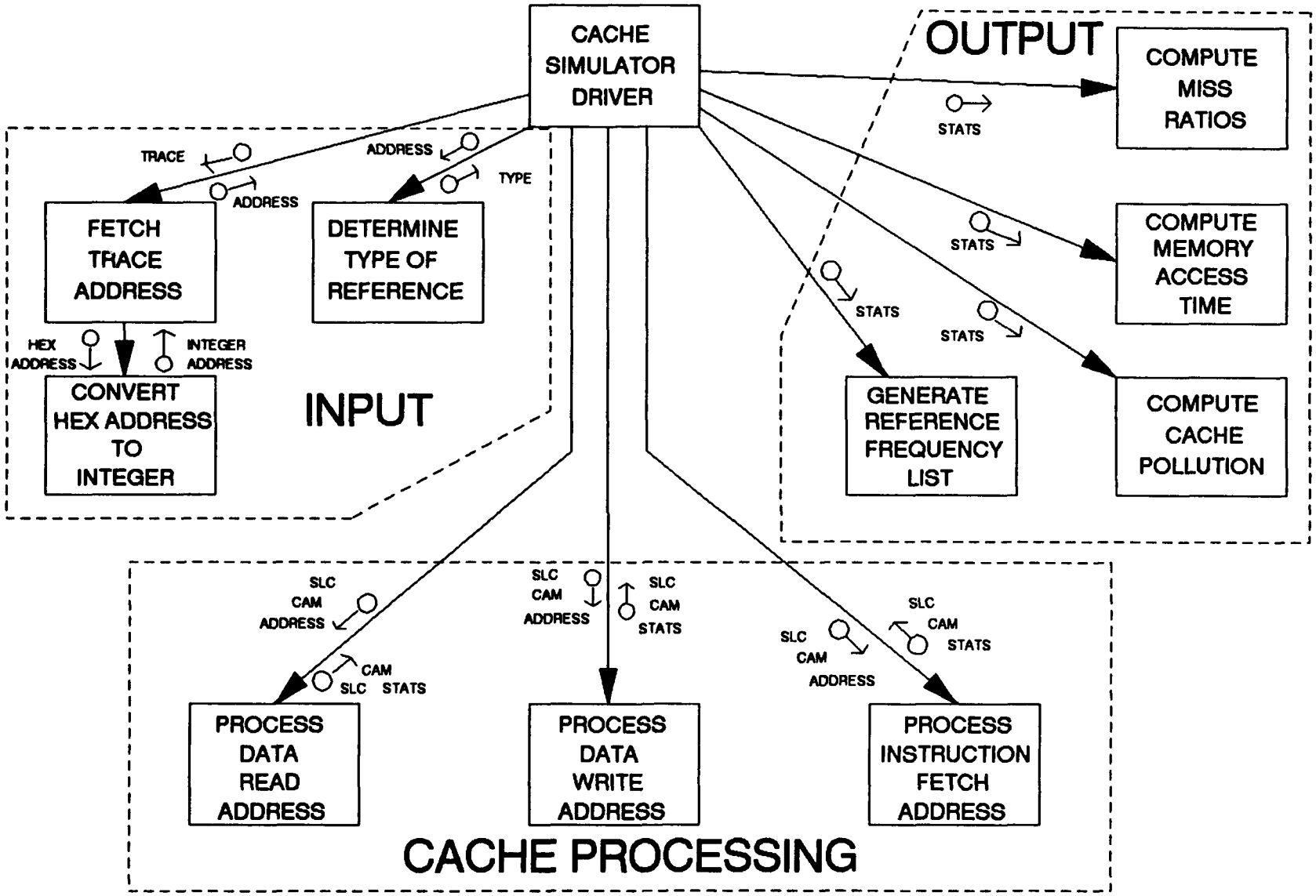



Figure 3.1: Cache Simulator Structure Chart

SLC and CAM cache functions, they require access to the SLC and CAM cache data structures. As the selected module processes the memory reference, SLC and CAM data is generated for cache performance analysis.

The output area processes the performance data and produces statistics which are written to files. This area is comprised of four modules. The *Compute Miss Ratios* module calculates the SLC and CAM cache miss ratios. Using the cache miss ratios, the *Compute Memory Access Time* module calculates the effective memory access time. The *Compute Cache Pollution* module calculates the SLC and CAM cache pollution percentages. And finally, the *Generate Reference Frequency List* module produces the output file tracking the number of references before each prefetched address is first referenced.

Table 3.1 maps the structure chart modules to the functional requirements satisfied.

## 3.4 Implementation of Cache Simulator in Ada

Once the simulator architecture was established, development of the cache simulator began. This development phase involved the transformation of the functional design requirements into a complete Ada system. Major tasks included the detailed design, coding, integration and testing of the simulator.

Table 3.1: Cache Simulator Requirements Matrix

| Module | Functional Requirements |
|---|---|
| Driver | 3.2.2 |
| Fetch Trace Address | 3.2.1 |
| Determine Reference Type | 3.2.1 |
| Convert Hex Address to Integer | 3.2.1 |
| Process Data Read Address | 3.2.2 |
| Process Data Write Address | 3.2.2 |
| Process Instruction Fetch Address | 3.2.2 |
| Compute Miss Ratios | 3.2.3 |
| Compute Memory Access Time | 3.2.4 |
| Compute Cache Pollution | 3.2.5 |
| Generate Reference Frequency List | 3.2.6 |

Each program module in the structure chart is mapped into an Ada package. By performing this encapsulation, each Ada package can be developed and tested as an individual functional component. Indicated by the arrow connections between modules (structure chart), visibility into other packages is accomplished by *withing* the required packages. Identified as data flows between modules (structure chart), package communication is accomplished through the passing of parameters to/from the various procedures and functions contained within the packages.

The following sections provide a detailed description of each Ada package used in the cache simulator. Appendix A contains the source code.

### 3.4.1 LinkedLists_Package

A linked list data structure is employed to represent the SLC cache abstract data type. The *LinkedLists_Package* is used to implement the SLC cache as a linked list structure. Ada code for this package is a modified version of code taken from Data Structures with Ada (Feldman, 1985:103-115). The rationale for the linked list structure is based on the LRU replacement algorithms of the SLC cache.

Initially, it appeared that the SLC might be optimally represented as an array. However, if an array was employed, the array elements would constantly have to be reshuffled to depict an LRU ordering. If a time stamp (number of references passed) was used, every array element would have to be searched to determine the LRU candidate for replacement. Both of these array approaches would have lead to simulator performance penalties.

A linked list structure provides a cleaner way to implement the LRU replacement algorithm. As an SLC address is being either referenced (cache hit) or prefetched, the address can simply be placed at the front of the list. As a result, the LRU addresses fall to the rear of the list. When an address has to be replaced, no search is required. The address at the rear of the list is replaced. In addition, since the SLC is very small, the size of the linked list never grows beyond 512 nodes (largest number of addresses in SLC). In turn, SLC searches do not degrade simulator performance.

In addition to the SLC, two linked lists are used to maintain the SLC and CAM reference frequency lists. These lists will be discussed in more detail later. For now, it should be pointed out that these two linked lists are handled by this package.

The *LinkedLists_Package* specification is shown in Figure 3.2. *LinkedListNode* is a record structure containing three fields. The *Addr* field represents the address of the memory reference. The *NumRef* field indicates the number of memory references which have passed while the address is waiting to be first referenced. The *Next* field contains the pointer to the next *LinkedListNode*.

This package has two functions and four procedures to manipulate the state of the linked list data structure. Receiving the memory reference address, the *MakeNode* function creates a *LinkedListNode* for storing a new address in the SLC. The *Search* function is used to find the node containing the matching address during a SLC search. The *AddToFront* procedure adds a *LinkedListNode* to the front of the SLC. The *Insert_In_Order* procedure is used by the SLC and CAM reference frequency lists to insert an address record in ascending order by the *NumRef* value. *Insert_In_Order* employs the *AddToFront* and *AddToRear* (only at end of list) procedures to place a frequency record in its appropriate position in the list. The *Delete* procedure deletes a designated node from the SLC.

```
package  LinkedLists_Package is

   type LinkedListNode;
   type NodePointer is access LinkedListNode;
   type LinkedListNode is
      record
         Addr     : integer;
         NumRef   : integer := 0;
         Next     : NodePointer := null;
      end record;

   type List is
      record
         Next     : NodePointer := null;
         Tail     : NodePointer := null;
      end record;

   function MakeNode (Address: integer)
                      return NodePointer;
   function Search (L: List; P1: NodePointer)
                      return NodePointer;
   procedure AddToFront (L: in out List;
                         P1: NodePointer);
   procedure AddToRear (L: in out List;
                        P1: NodePointer);
   procedure Insert_In_Order (L: in out List;
                              P1: NodePointer);
   procedure Delete (L: in out List; P1: NodePointer);

end LinkedLists_Package;
```

Figure 3.2: *LinkedList_Package* Specification

The *LinkedLists_Package* body instantiates a generic package called *Unchecked_Deallocation*. Dynamic allocation can swiftly use up storage space as linked lists nodes are continually being created and deleted. The *Unchecked_Deallocation* package is called by the *Delete* procedure to return space used by the deleted node back to available memory.

### 3.4.2 CircularQ_Package

A circular queue data structure is used to implement the FIFO circular buffer replacement algorithm of the CAM cache. Blocks of addresses are stored in the CAM in the order they are referenced. The *CircularQ_Package* maintains the state of the CAM cache using the circular queue. Ada code for this package is also a modified version of code taken from Feldman (Feldman, 1985:144-145).

Using a circular queue eliminates the need to move the entire queue. Since the front and rear of the queue are essentially connected, CAM replacement and prefetch operations can be accomplished by moving head and tail pointers around the queue.

In Chapter 4, the CAM is referred to as a "*stack*." This terminology is not to be confused with the CAM being implemented as a circular queue. Instead, "*stack*" is used to describe the memory referencing behavior of the CAM cache: *same-stack-distance*, *not-same-stack-distance*. The circular queue describes how the addresses are stored and replaced within the CAM cache.

The *CircularQ_Package* specification is shown in Figure 3.3. The circular queue array is created as the dynamically allocated *Array_Type* with index of range 1 to 32768. The upper range represents the largest possible CAM size of 32768 addresses. The *Queue* is a record structure containing four fields. Declared as a dynamically allocated

```
package  CircularQ_Package is

  type index is range 1 .. 32768;
  type Array_Type is array (index) of integer;
  type Array_Ptr_Type is access Array_Type;

  type Queue is
    record
       Address   : Array_Ptr_Type := new Array_Type;
       Ref_Count : Array_Ptr_Type := new Array_Type;
       head      : index;
       tail      : index;
    end record;

  procedure Enqueue (Q: in out Queue;
                     CAM_Size_Index: in index;
                     Reference: in integer);

  procedure Dequeue (Q: in out Queue;
                     CAM_Size_Index: in index);

  procedure SearchQ (Q: in Queue;
                     Reference: in integer;
                     CAM_Size_Index: in index;
                     Position: in out index;
                     Found: in out boolean);

end CircularQ_Package;
```

Figure 3.3: *CircularQ_Package* Specification

variable of *Array_Ptr_Type*, the *Address* field is used to store the memory reference address in the CAM. Also of *Array_Ptr_Type*, the *Ref_Count* field indicates the number of memory references that have passed while the address is waiting to be first referenced. The *head* and *tail* fields are used to indicate the front and rear of the CAM circular queue, respectively.

This package has three procedures to manipulate the state

of the circular queue data structure. Receiving the prefetched memory reference address, the *Enqueue* procedure loads the address at the tail of CAM queue. As required by the FIFO replacement algorithm, the *Dequeue* procedure removes the address located at the head of the CAM queue. The *SearchQ* procedure searches for a matching address in the CAM. It returns a flag indicating whether or not an address has been found. If a cache hit occurs, *SearchQ* will also return the position of the address in the CAM queue.

### 3.4.3 Addr_Record_Package

For type handling purposes, an abstract data type is created for the virtual addresses contained in the Agarwal, Sites, and Horowitz trace files. The *Address_Record_Package* specification is shown in figure 3.4. This package creates a record structure comprised of two fields. The *The_Type* field indicates the type of the memory reference: data read, data write, or instruction fetch. The *Address* field stores the hexadecimal virtual address from the trace file.

### 3.4.4 Cache_Simulator Driver Procedure

The *Cache_Simulator* driver procedure functions as the main controller for the simulator system. It provides an interface between the input, cache processing, and output packages. In order to produce an executable Ada system, the *Cache_Simulator* driver is built as a procedure instead of as

```
package  Addr_Record_Package  is

   type Addr_Record is
      record
         The_Type  : character;
         Address   : integer;
      end record;

end Addr_Record_Package;
```

Figure 3.4: *Addr_Record_Package* Specification

a package. Figure 3.5 shows the *Cache_Simulator* driver procedure.

As the driver, the *Cache_Simulator* procedure needs visibility to the input, processing, output packages. The required packages are *withed* into the driver.

In the variable declaration section, the SLC linked list and CAM circular queue data structures are instantiated. The SLC and CAM reference frequency lists are also instantiated. The SLC size, CAM size, SLC block size, and CAM block size parameters are declared as variables rather than constants. This allows the cache parameters to be interactively entered by the user. The variables used to calculate the cache miss ratios, pollution, and effective memory access time are declared and initialized to zero. These variables include the following: *SLC_Miss*, *CAM_Miss*, *SLC_Total_Refs*, *CAM_Total_Refs*, *SLC_Non_Ref*, *CAM_Non_Ref*, *SLC_Total_Prefetch*, and *CAM_Total_Prefetch*.

For convenience, the driver provides the user with an

```
with Text_IO, Addr_Record_Package, LinkedLists_Package,
     CircularQ_Package, Fetch_Address_Package,
     Determine_Type_Package, Serv_Instr_Fetch_Package,
     Serv_Data_Read_Package, Serv_Data_Write_Package,
     Compute_Miss_Ratios_Package,
     Compute_Memory_Access_Time_Package,
     Compute_Cache_Pollution_Package,
     Generate_Ref_Frequency_List_Package;

procedure  Cache_Simulator is
   --  ********************************
   --  * Variable declaration section *
   --  ********************************
begin
   --  user enters names of output & reference files
   --  create output & reference files
   --  user enter name of trace input file
   --  open trace input file
   --  user enters cache parameters:  SLC size, CAM size,
   --     SLC block size, CAM block size
   --
   --  while not end of file loop
   --     call routine to fetch address from trace file
   --     call routine to determine type of reference
   --     case
   --        when data read =>
   --           call routine to process data read
   --        when data write =>
   --           call routine to process data write
   --        when instruction fetch =>
   --           call routine to process instruction fetch
   --        when others => exit
   --     end case
   --     if number of references processed = 20000 then
   --        call routine to compute miss ratios
   --        reset reference counter to zero
   --     else if end of trace file then
   --        call routine to compute final miss ratios
   --        call routine to compute average memory
   --           access time
   --        call routine to compute cache pollution
   --        call routine to generate reference
   --           frequency list
   --     end if
   --  end loop
   --  close input, output, and reference files

end Cache_Simulator
```

Figure 3.5: *Cache_Simulator Driver* Procedure

interface for executing a simulation. Once the simulator is activated, the user is queried to name the statistics output file to be generated by the simulation run. The statistics output file includes the SLC and CAM miss ratios, the SLC and CAM pollution percentages, and the effective memory access time. Secondly, the user is asked to name the reference output file. This file includes the reference frequency list for the SLC and CAM caches. Next, the user is queried to name the trace input file to be used in the simulation run. The final user inputs include the cache parameters: SLC size, CAM size, SLC block size, and CAM block size. For archival purposes, the driver writes the trace input filename and the cache parameters at the top of the statistics and reference output files.

Once all user inputs have been entered, the cache simulation begins. Due to the different formats of the address traces, two Cache_Simulator drivers have been written. One version is designed for the Agarwal, Sites, and Horowitz traces. In this version (*Version 1*), the driver calls the *Load_Record* procedure (*Fetch_Address_Package*). This action returns an address converted from an hexadecimal to an integer format. The driver then calls the *Address_Type* function (*Determine_Type_Package*) to determine the type of reference.

The other driver version (*Version 2*) is designed for the Explorer trace files. As previously explained, these address traces are already in integer format and do not require a type

determination. In turn, this driver version only needs to perform a *get (Text_IO)* operation to fetch an address.

Depending on the reference type, the *Version 1* driver will invoke one of the three cache processing procedures: *Serv_Data_Read*, *Serv_Data_Write*, or *Serv_Instr_Fetch*. For this research, these three procedures perform the same cache processing functions. Given this, the rationale for creating three separate procedures is to accommodate future research requirements. Continuing research may need the simulator to perform different cache processing functions based on reference type. Separate procedures facilitate implementation of these future cache processing modifications.

Since no reference type determination occurs, the *Version 2* driver only requires one main cache processing procedure. The *Process_Memory_Reference* procedure performs the same cache functions as the three processing procedures in the *Version 1* driver.

Once the memory address reference has been processed, the driver repeats the fetch and process cycle described above. For every 20000 trace references processed, the driver calls the *Compute_Miss_Ratios* procedure to calculate the cumulative miss ratios in the SLC and CAM caches. When the simulator reaches the end of the trace file, the driver invokes several procedures to produce the cache performance data. First, the driver calls the *Compute_Miss_Ratios* procedure to calculate the final miss ratios. Next, the driver calls the

*Compute_Memory_Access_Time* procedure to calculate the effective memory access time. Then, the *Compute_Cache_Pollution* procedure is called to determine the pollution percentages. And finally, the *Generate_Ref_Frequency_List* procedure is invoked producing the reference frequency data.

At this point, the simulation is finished. The driver concludes the session by closing the trace, statistics, and reference files.

### 3.4.5 Cache Processing Packages

The cache processing packages include the following: (Version 1 driver) *Serv_Data_Read_Package*, *Serv_Data_Write_Package*, and *Serv_Instr_Fetch_Package*; (Version 2 driver) *Process_Memory_Reference_Package*. Since all four packages perform the same cache processing functions, the following explanation applies to all four packages.

For types handling, the cache processing packages require visibility (*with*) to the *Linked_List_Package*, the *CircularQ_Package*, and the *Addr_Record_Package*. Each package contains one cache processing procedure. To assist in a more detailed description, Figure 3.6 shows the flow of the cache processing procedure.

Using the requested address, the SLC is searched for a matching address. If a SLC hit occurs, then the SLC performance statistics are updated. Since the address request has been satisfied, the cache processing is finished. Control

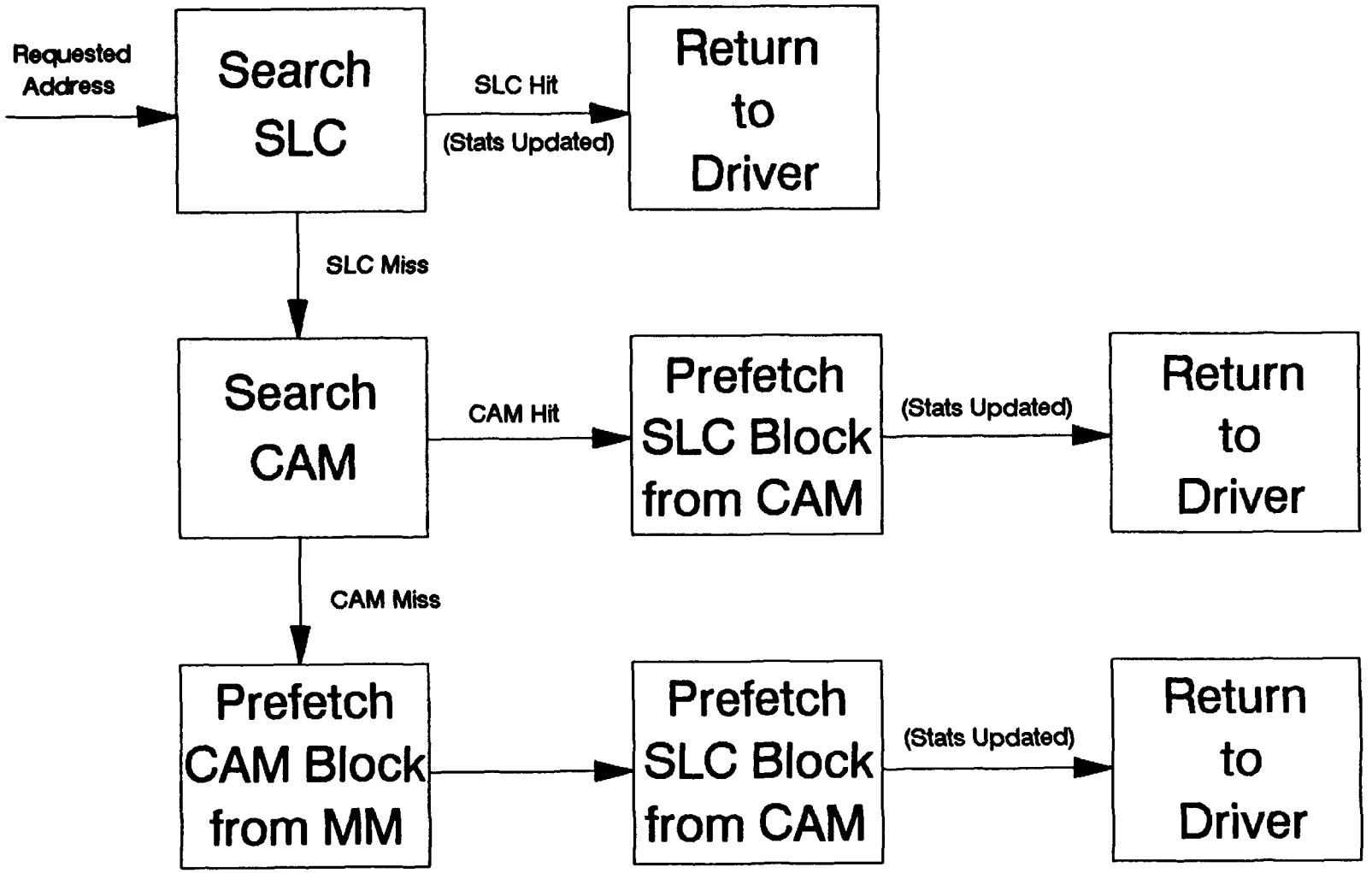



Figure 3.6: Cache Processing Flow

is returned to the driver.

If a SLC miss occurs, then the CAM is searched for a matching address. If a CAM hit occurs, then the block containing the matching address is prefetched from the CAM to the SLC. Figure 3.7 illustrates this prefetching process. The goal is to prefetch the structural locality that exists in the CAM. In turn, the requested address plus the addresses located immediately above in the CAM (equal to SLC block size) are prefetched into the SLC. The prefetched addresses are placed at the front of the SLC linked list. This action causes the LRU addresses to fall toward the back of the list. Once the SLC prefetch is accomplished, the CAM performance statistics are updated. Since cache processing has finished, control is returned to the driver.

If a CAM miss occurs, then the block containing the requested address is prefetched from main memory to the CAM. In addition, a SLC block containing the requested address must be prefetched from the CAM. Figure 3.8 illustrates this prefetching process. Once again, only the requested address plus the addresses immediately above in the stack (equal to SLC block size) are prefetched into the SLC. In the example, although the SLC block size is four, the prefetch results in only one word. Two conditions produce this result. One is the CAM prefetch is always placed at the stack top. The other is the requested address is the last address on the stack. Once both prefetches are accomplished, the SLC and CAM

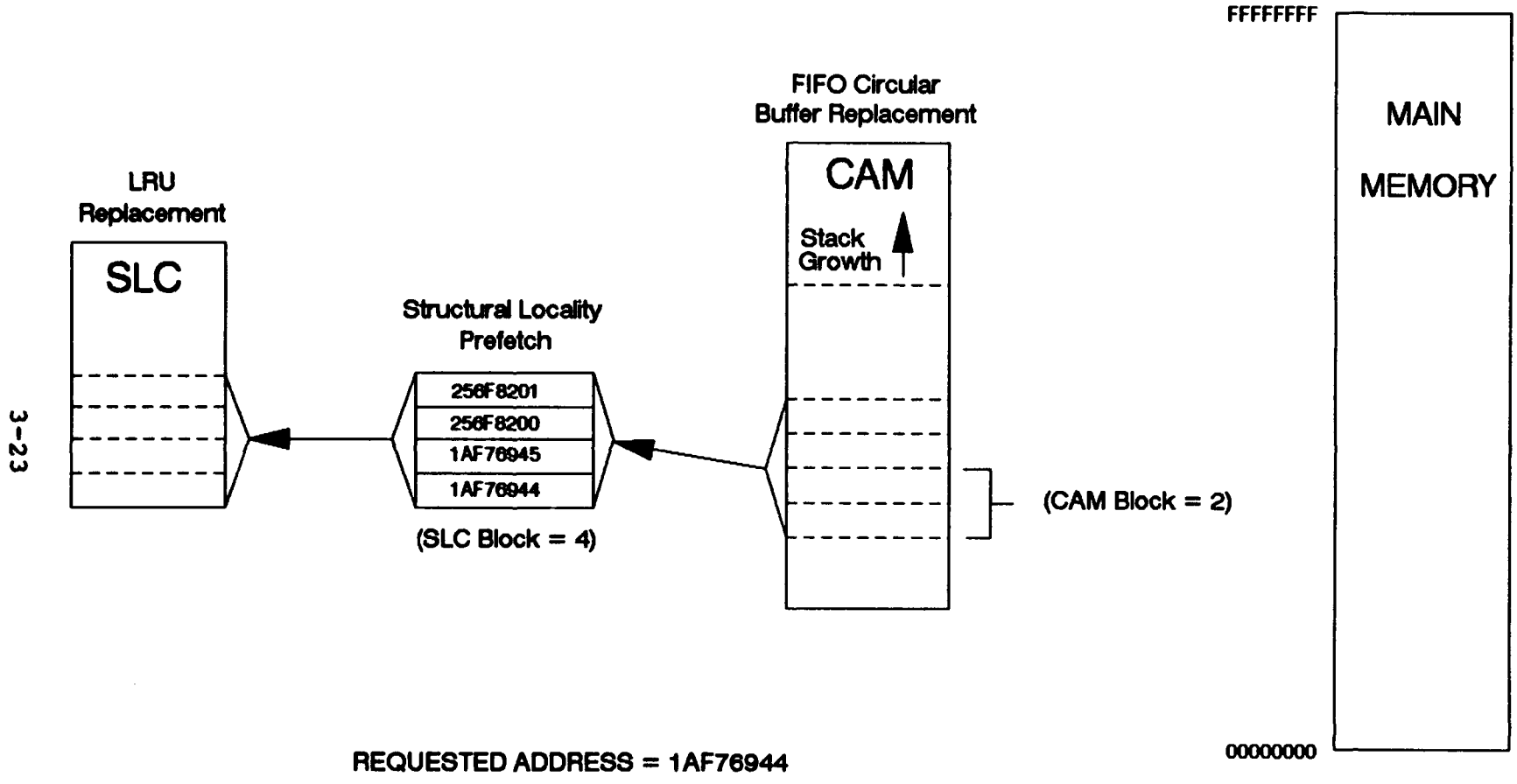



Figure 3.7: SLC Prefetch After SLC Miss & CAM Hit

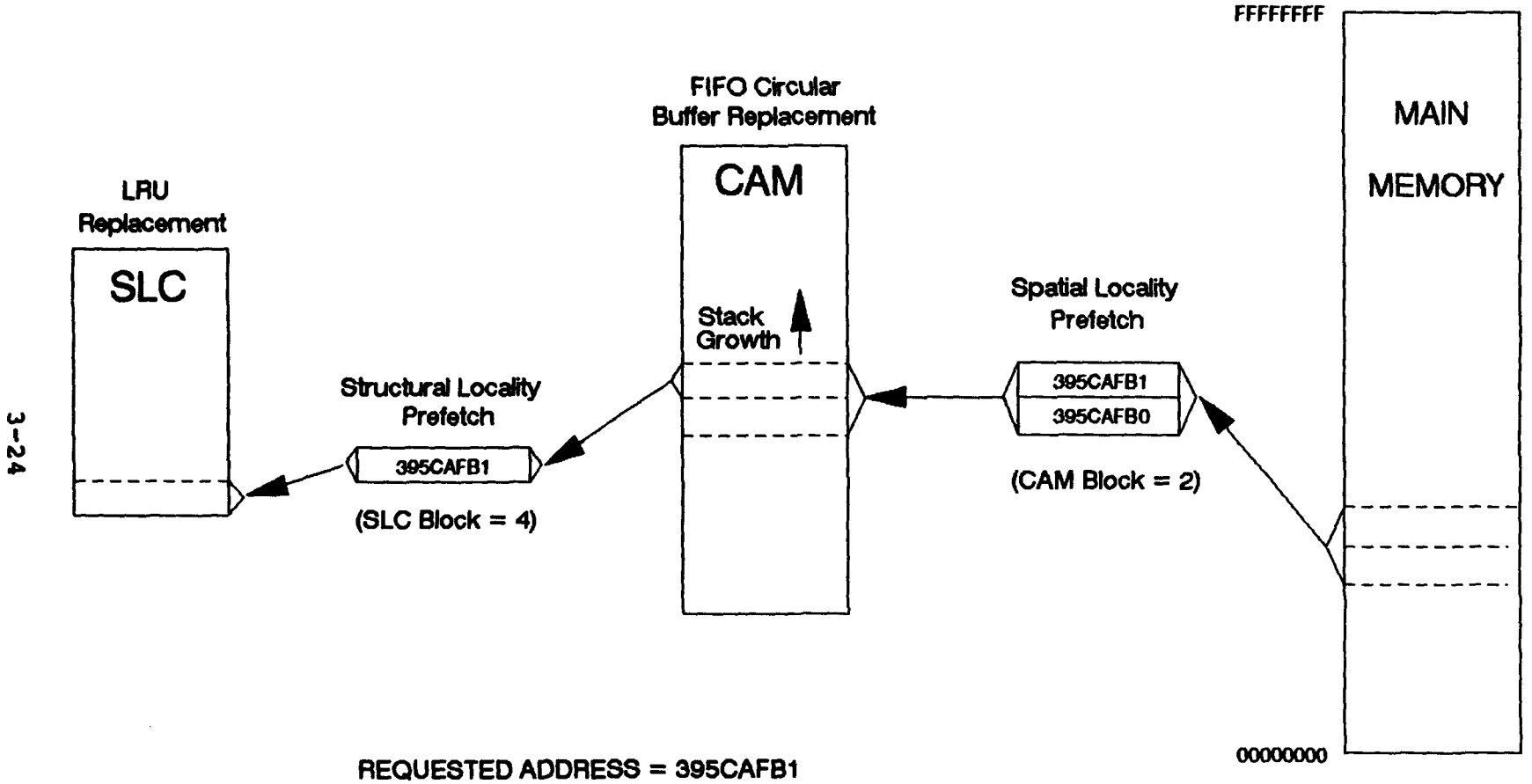



Figure 3.8: SLC & CAM Prefetches After Both Caches Missed

performance statistics are updated. Since cache processing has finished, control is returned to the driver.

### 3.4.6 Fetch_Address_Package

The *Fetch_Address_Package* contains one procedure: *Load_Record*. The specification for this package is shown in Figure 3.9. *Load_Record* procedure extracts one memory reference record (address and reference type) from the current position of the Agarwal, Sites, and Horowitz trace file. To convert the address from hexadecimal to integer format, this procedure calls the *Hex_to_Dec* procedure. The resulting integer address record is then returned to the driver for further processing.

```
with  Text_IO, Addr_Record_Package;

package  Fetch_Address_Package  is

   procedure Load_Record
             (Input_File: in out File_Type;
              Memory_Ref: out Addr_Record);

end Fetch_Address_Package;
```

Figure 3.9: *Fetch_Address_Package* Specification

### 3.4.7 Hex_to_Dec_Package

The *Hex_to_Dec_Package* contains one function: *Hex_to_Dec*. The specification for this package is shown in

Figure 3.10. The *Hex_to_Dec* function converts the input hexadecimal string into an integer address.

Due to the address structure of the SPARC microprocessor, an offset is required in the conversion value. Although the SPARC machine has a 32-bit address, it reserves 1 bit as a sign bit for integers. In turn, not all eight digit hexadecimal addresses can be represented as integer values. To counter this, the integers are offset to include both the positive and negative values. The resulting integer range is: $-2^{31}$ .. $(2^{31} - 1)$. Although this offset changes the original value of the virtual address, the modified value does not affect the cache processing. The simulator is only interested in matching addresses during searches.

The rationale for converting the hexadecimal address to an integer value is to optimize performance. The simulator

```
package  Hex_to_Dec_Package  is

   function  Hex_to_Dec  (Hex_Addr: string;
        Hex_Length: natural) return integer;

end  Hex_to_Dec_Package;
```

Figure 3.10: *Hex_to_Dec_Package* Specification

could be designed to process a hexadecimal address. However, since the address would be represented as a string, cache searches would involve comparing addresses one character at a time. The result would be a substantial drop in simulation

speed. By treating the address as an integer value, cache searches only require a single comparison per address.

### 3.4.8 Determine_Type_Package

The *Determine_Type_Package* contains one function: *Address_Type*. The specification for this package is shown is Figure 3.11. Receiving the address record, the *Address_Type* procedure determines the memory reference type: data read, data write, or instruction fetch. The type identifier is returned to the driver.

```
with Addr_Record_Package;

package  Determine_Type_Package  is

   function Address_Type (Memory_Ref: Addr_Record)
                          return character;

end Determine_Type_Package;
```

Figure 3.11: *Determine_Type_Package* Specification

### 3.4.9 Compute_Miss_Ratios_Package

The *Compute_Miss_Ratios_Package* contains one procedure: *Compute_Miss_Ratios*. The specification for this package is shown in Figure 3.12.

The *Compute_Miss_Ratios* procedure calculates the SLC and CAM miss ratios using input cache performance statistics: number of SLC misses, total number of SLC references, number

of CAM misses, and total number of CAM references. The resulting miss ratios are written to the statistics file. In addition, the values are returned to the driver to be used in effective memory access time calculations.

```
with  Text_IO;

package  Compute_Miss_Ratios_Package  is

   procedure Compute_Miss_Ratios
      (SLC_Miss: in natural; CAM_Miss: in natural;
       SLC_Total_Refs: in natural;
       CAM_Total_Refs: in natural;
       Num_Ref: in natural;
       SLC_MR: out float; CAM_MR: out float
       Output_File: in out File_Type);

end Compute_Miss_Ratios_Package;
```

Figure 3.12: *Compute_Miss_Ratios_Package* Specification

### 3.4.10 Compute_Memory_Access_Time_Package

The *Compute_Memory_Access_Time_Package* contains one procedure: *Compute_Memory_Access_Time*. The specification for this package is shown in Figure 3.13. The Compute_Memory_Access_Time procedure calculates the effective memory access time for the trace workload. The access times (in clock cycles) used in the model for the three memory levels are as follows: SLC: 1; CAM: 4; main memory: 32.

```
with Text_IO;

package  Compute_Memory_Access_Time_Package  is

   procedure Compute_Memory_Access_Time
      (SLC_MR: in float; CAM_MR: in float;
       Output_File: in out File_Type);

end Compute_Memory_Access_Time_Package;
```

Figure 3.13: *Compute_Memory_Access_Time_Package* Specification

### 3.4.11 Compute_Cache_Pollution_Package

The *Compute_Cache_Pollution_Package* contains one procedure: *Compute_Cache_Pollution*. The specification for this package is shown in Figure 3.14. The *Compute_Cache_Pollution* procedure calculates the SLC and CAM cache pollution using the

```
with  Text_IO, LinkedLists_Package, CircularQ_Package;

package  Compute_Cache_Pollution_Package  is

   procedure Compute_Cache_Pollution
             (SLC: in out List; CAM in out Queue;
              SLC_Non_Ref: in out natural;
              CAM_Non_Ref: in out natural;
              SLC_Total_Prefetch: in natural;
              CAM_Total_Prefetch: in natural;
              Temp_CAM_Size: in natural;
              Output_File: in out File_Type);

end Compute_Cache_Pollution_Package;
```

Figure 3.14: *Compute_Cache_Pollution_Package* Specification

following cache performance statistics: number of prefetched

addresses in the SLC and CAM never referenced; total number of prefetched addresses in the SLC and CAM. The resulting pollution percentages are written to the statistics file.

### 3.4.12 Generate_Ref_Frequency_List_Package

The *Generate_Ref_Frequency_List_Package* contains one procedure: *Generate_Ref_Frequency_List*. The specification for this package is shown in Figure 3.15.

The *Generate_Ref_Frequency_List* procedure produces the output file recording the reference frequencies for the SLC and CAM. The file format consists of two columns. The first column is comprised of values denoting the number of references passed before the prefetched addresses were first referenced. The second column indicates the number of addresses (frequency) which realized the corresponding number of references value.

```
with Text_IO, LinkedLists_Package;

package  Generate_Ref_Frequency_List_Package  is

   procedure Generate_Ref_Frequency_List
             (SLC_Ref_List: in out List;
              CAM_Ref_List: in out List;
              SLC_Non_Ref: in natural;
              CAM_Non_Ref: in natural;
              Reference_File: in out File_Type);

end Generate_Ref_Frequency_List_Package;
```

Figure 3.15: Generate_Ref_Frequency_List_Package Spec

## 3.5 Validation of Cache Simulator

Prior to being integrated into the cache simulator system, each Ada package was tested thoroughly as a unit. Using test inputs, the outputs of each package were checked for correctness. Verifying the correct functioning of the packages facilitated the integration effort.

Once the integration of the Ada packages was successfully completed, the cache simulator system was ready for testing. In order to ensure the validity of the research results, the simulator was extensively tested for accuracy. To accomplish this, a test trace file was created. The file contained 50 memory references which would require all cache processing functions to be used. Figure 3.16 provides an outline of the testing requirements. Using this trace file, the test simulations were designed to meet all of these requirements. In Appendix B, each test requirement is mapped to the point within the trace where it is tested.

Five test trace simulations were run and checked for accuracy. Each test run used a different set of cache parameters: SLC size, SLC block size, CAM size, and CAM block size. Prior to running the simulations, the cache performance statistics were manually calculated for each set of cache parameters. After the five test simulations were run, the simulation statistics matched the manual calculations.

I. Test Input Functions
(Version 1 driver)
A. Correct handling of memory reference record
1. Hexadecimal address (8 digits)
2. Hexadecimal address (< 8 digits)
3. Conversion to integer value
4. Type determination
(Version 2 driver)
B. Correct handling of integer memory reference
C. (Both) Able to process all memory references

II. Test Cache Processing Functions
A. Correct handling of SLC search
1. SLC hit
2. SLC miss
B. Correct handling of CAM search
1. CAM hit
2. CAM miss
C. On CAM hit, correct handling of SLC prefetch
1. Before SLC fills
2. After SLC fills
3. From middle of CAM
4. From top of CAM
D. On CAM miss, correct handling of CAM prefetch and SLC Prefetch
1. Before CAM fills
2. After CAM fills
3. Before SLC fills
4. After SLC fills
E. Correct tracking of cache performance stats (after 25 references)
1. # of SLC misses
2. # of CAM misses
3. total # of SLC references
4. total # of CAM references

III. Test Output Functions
A. Correct SLC & CAM miss rates
B. Correct effective memory access time
C. Correct SLC & CAM cache pollution percentages
D. Correct values in reference frequency file

Figure 3.16: Simulator Testing Requirements

## 3.6 Summary

The cache simulator was designed to meet all the functional requirements of the proposed memory subsystem. Once developed, the simulator was subjected to rigorous validation testing and cross-checking. Based on this effort, it can be concluded that the cache simulator is capable of providing valid results for this research.

# Chapter 4

# Findings

This chapter provides the research results of the effect that spatial locality prefetching has on structural locality. First, the workload selection for studying the cache pollution in the SLC and CAM is discussed. Next, the modifications to Hobart's memory referencing models are shown to account for spatial prefetching. From these modified models, new equations are derived to predict the SLC and CAM hit probabilities. Next, the approach to using simulation measurements to solve these equations is explained. Results from these equations are then compared against hit ratios produced in the trace-driven simulations. The SLC and CAM hit probabilities are then used to estimate the effective memory access time of the design. Finally, the performance effects of spatial prefetching are compared with baseline results using no prefetching.

## 4.1 Workload Selection

As discussed in chapter two, the differences between the memory referencing behavior of symbolic and conventional workloads has been well documented. In order to produce meaningful results, cache performance studies must take these

differences into account.

The basic motivation for the proposed memory subsystem was to exploit the unique locality characteristics of symbolic workloads (Hobart, 1989:96). As such, this research focuses on symbolic workloads in determining the effects of cache pollution from spatial prefetching. The symbolic workloads used in the trace-driven simulations are shown in Table 4.1. Eight workloads were used from the Explorer traces. The last workload was taken from the VAX traces.

Table 4.1: Symbolic Workloads Used in Simulations

| Workload | Application | System |
|---|---|---|
| Boyer | Theorem Prover | Explorer II |
| Compile-RB | Lisp Compiler | " |
| Compile-STR | Lisp Compiler | " |
| GLISP-Comp | Expert System Tool | " |
| GLISP-Pay | Expert System Tool | " |
| QSIM | Qualitative Reasoning | " |
| Reducer | Symbolic Computation | " |
| TMYCIN | Expert System Tool | " |
| LISP.000.DIN | Lisp Application | VAX 8200 |

The nine selected workloads included all references. The rationale is to characterize the cache pollution effects of spatial prefetching against total workload behavior.

## 4.2 Trace-driven Simulations

To reasonably limit the number of simulations due to time constraints, the selection of the SLC and CAM size parameters is based on an estimated upper range of current cache technology. The CAM and SLC sizes are set at 8192 and 512 words, respectively. This 16 to 1 size ratio meets the minimum 8 to 1 ratio requirement. The cache parameters chosen for these simulations are shown in Table 4.2.

In order to study the cache pollution effects of spatial prefetching in the CAM, the CAM block size is fixed at 4 words. The rationale for this block size choice was based on the availability of data needed to solve the cache hit probability equations.

In studying cache performance in a RISC environment, Hill and Pnevmatikatos determined that a 32 byte block size (upper limit) produced the lowest miss ratios for a cache size of 32K

Table 4.2: SLC and CAM Cache Parameters

| Type | Number of Words |
|---|---|
| SLC Size | 512 |
| CAM Size | 8192 |
| SLC Block Size | 4, 8, 16, 32 |
| CAM Block Size | 4 |

bytes (Hill and Pnev, 1990:53-68). The block size of four words (used for the CAM cache) was also found to produce low miss ratios. Using trace-driven simulation in their research, the effects of block sizes were analyzed using a variety of workloads including "*Xlisp*" (lisp interpreter with object-oriented features).

The three SLC block sizes represent 1:1, 2:1, 4:1, and 8:1 ratios to the CAM block size. The rationale for these block size selections is based on the memory subsystem design. The CAM is designed to capture the structural locality inherent in the workloads while spatially prefetching. In order to take advantage of this structural locality, the SLC should prefetch a multiple of the CAM block size.

The speed performance of the cache simulator is very acceptable. Assuming one user on the SPARC workstation, simulation run time varied from one half hour to two hours. This run time reflects simulations using trace files of up to 450,000 references. Importantly, this research showed the cache simulator could be structurally developed using the Ada language and, in turn, still provide a high level of performance.

A total of 45 simulations (five per workload) were run. Thirty-six simulations involved the prefetch ratios described above. The other nine used the same SLC and CAM sizes but did not use any prefetching. Therefore, the SLC and CAM block sizes were one word. The results from these simulations were

used as a baseline to determine any performance improvements.

The cache performance statistics generated from the trace simulations are shown in Table 4.3. The statistics represent the mean values for all workloads.

Table 4.3: Cache Performance Statistics

SLC Size = 512 CAM Size = 8192
SLC Block Size (varied) CAM Block Size = 4
No Prefetch: SLC & CAM Block Sizes = 1

| SLC Block: | 4 | 8 | 16 | 32 | No Prefetch |
|---|---|---|---|---|---|
| SLC Hit Rate | | | | | |
| Mean: | .859 | .880 | .893 | .908 | .820 |
| Std Dev: | .061 | .055 | .051 | .044 | .074 |
| CAM Hit Rate (given SLC miss) | | | | | |
| Mean: | .855 | .827 | .802 | .777 | .754 |
| Std Dev: | .063 | .080 | .100 | .106 | .113 |
| SLC Pollution | .608 | .660 | .733 | .807 | .489 |
| CAM Pollution (SLC hits counted as pollution in CAM) | | | | | |
| | .771 | .799 | .819 | .828 | .561 |
| Eff Memory Access Time* | 4.93 | 4.85 | 4.79 | 4.67 | 5.25 |

* The equation for effective memory access time was explained in Chapter 3 (Section 3.2.4).

## 4.3 CAM Cache Hit Probability

Hobart developed a two-state Markov model to illustrate CAM cache referencing when prefetching is used (Hobart, 1989:100-106). As shown in Figure 4.1, the model uses state transitions to represent the probabilities of various types of

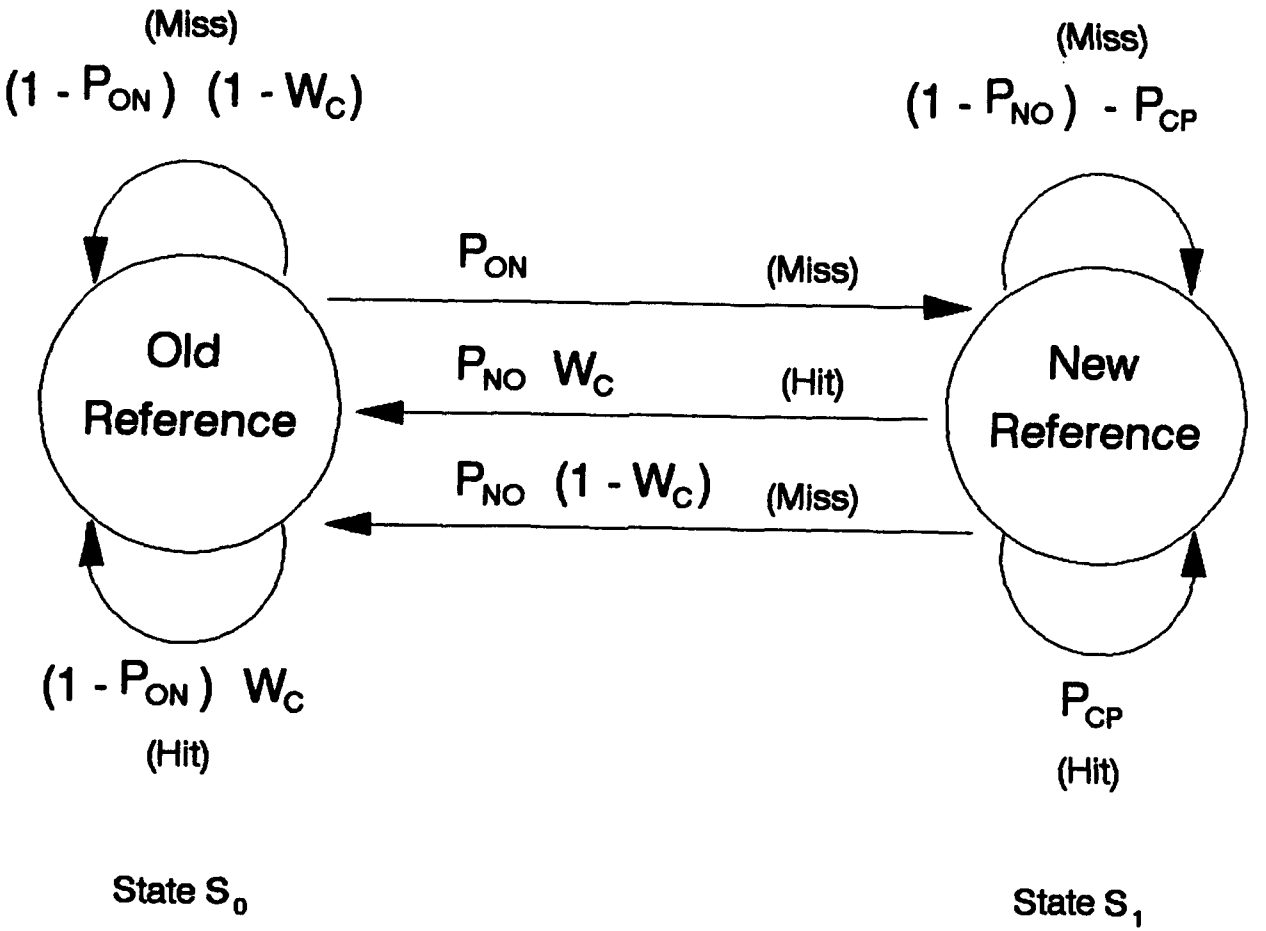



Figure 4.1: Markov Model for CAM Cache Referencing with Prefetching

CAM cache referencing behavior.

In this research, the CAM is assumed to be referenced only when a SLC miss occurs. Thus, the CAM miss rate represents the local miss rate of the CAM. From this assumption, a CAM hit can occur in three ways. Given an old reference is currently being referenced, a hit can take place if the next reference is also old and has not been replaced out of the CAM: $(1 - P_{ON})\ W_C$ where $(1 - P_{ON})$ is the probability of an old to old state transition; $W_C$ is the effective cache size hit probability. Given a current new reference, a hit occurs if the next reference is old and has not been replaced out of the CAM: $P_{ON}\ W_C$ where $P_{NO}$ is the probability of a new to old state transition. In addition, given a current new reference, a hit can take place if the next reference is new and exists within the same block: $P_{CP}$ which is the probability that a new to new reference is made to the block that was just prefetched. The simplifying assumption is made that once consecutive new references are made to a block and a new to old state transition occurs, any unreferenced addresses in the prefetched block(s) are considered pollution.

Thus, in the original model, when an old to new state transition occurs, the new reference is assumed to be to a previously unreferenced block. In order to more accurately predict the effects of spatial prefetching, a new model removes this assumption as shown in Figure 4.2 (modifications within dashed box). Given a current old reference, a hit can

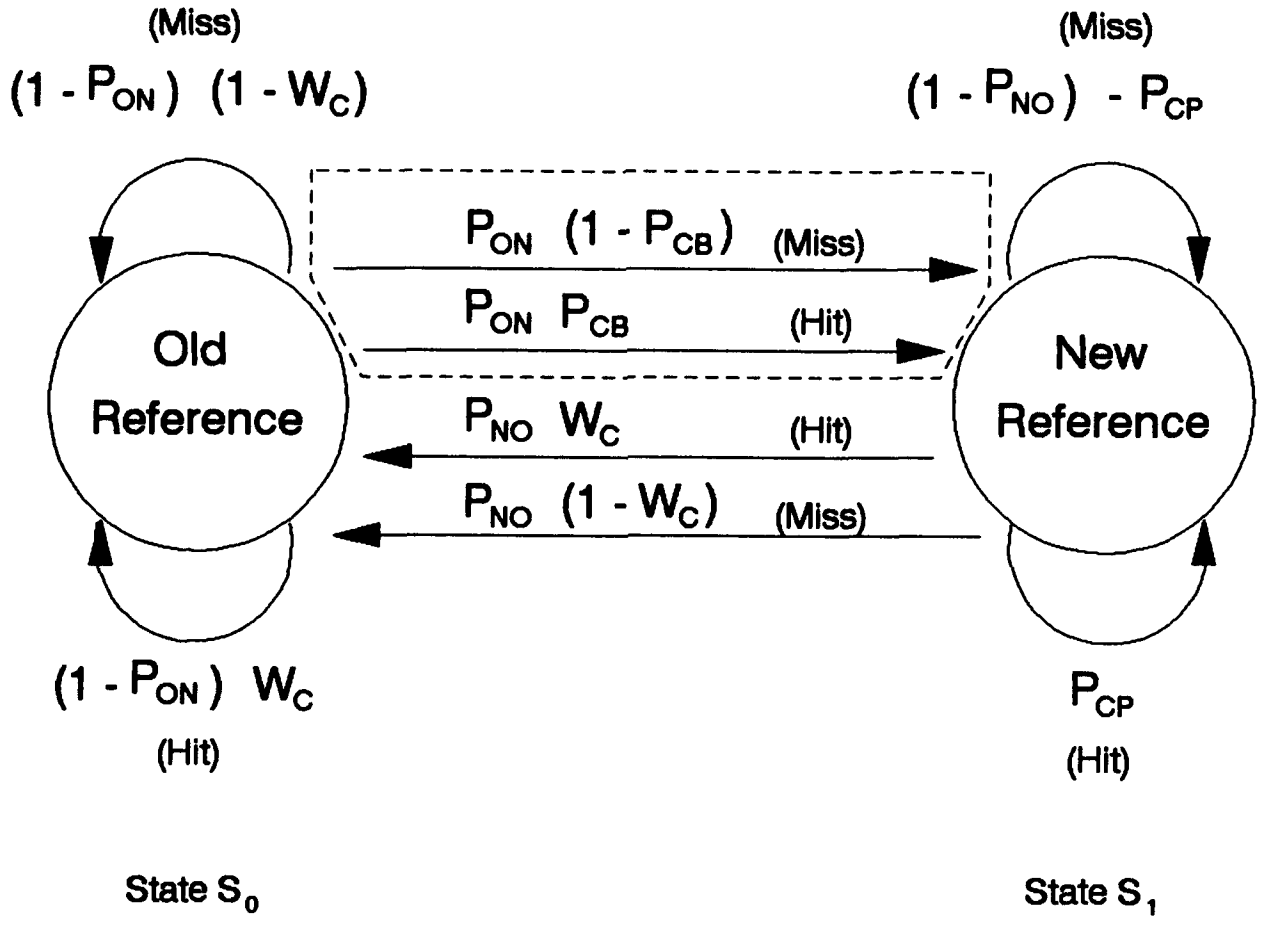



Figure 4.2: Modified Markov Model for CAM Cache Referencing with Spatial Locality Prefetching

take place if the next reference is new and is located within an existing prefetched block: $P_{ON}\ P_{CB}$ where $P_{CB}$ is the probability of referencing a CAM prefetched block.

Using the modified Markov model, the following equation is derived to determine the expected CAM cache hit rate, $P_{CAM}$:

$$P_{CAM} = ((1 - P_{ON})\, W_C + P_{ON}P_{CB})\ p_{s_0} + (P_{NO}W_C + P_{CP})\ p_{s_1}$$

where

$$p_{s_0} = \frac{P_{NO}}{P_{NO} + P_{ON}}$$

and

$$p_{s_1} = 1 - p_{s_0} = \frac{P_{ON}}{P_{NO} + P_{ON}}$$

Simplifying the equation, we have:

$$P_{CAM} = \frac{P_{NO}(W_C + P_{ON}P_{CB}) + P_{ON}P_{CP}}{P_{NO} + P_{ON}}$$

In order to show how the $P_{CAM}$ equation is solved, an example using the performance statistics (Table 4.3) involving a CAM block size of 4 words is explained.

By analyzing the temporal distances of reference strings, Hobart determined the state transition probabilities of memory referencing behavior for the symbolic trace workloads (Hobart, 1989:40-41, 137). The mean state transition probabilities (shown in Table 4.4) will be used in the hit probability equation.

Table 4.4: State Transition Probabilities - All References

| Workload | New-Old (NO) | Same Stack Distance (SSD) | Not Same Stack Distance (NSSD) | Old-New (ON) |
|---|---|---|---|---|
| Boyer | .506 | .474 | .502 | .024 |
| Comp-RB | .382 | .562 | .423 | .015 |
| Comp-STR | .383 | .544 | .438 | .018 |
| GLISP-C | .478 | .623 | .361 | .016 |
| GLISP-P | .250 | .588 | .407 | .005 |
| QSIM | .457 | .444 | .544 | .012 |
| Reducer | .137 | .540 | .454 | .006 |
| TMYCIN | .378 | .626 | .364 | .010 |
| Mean | .371 | .550 | .436 | .013 |
| Std Dev | .1235 | .0653 | .0634 | .0063 |

$W_C$ is a function of the CAM size. The probability that an old reference exists in the CAM is limited by the finite size of the CAM. Since spatial prefetching results in pollution, the effective size of the CAM is reduced. Effective CAM size is calculated as follows:

$$Eff\ CAM\ Size = N_C\ (1 - C_p)$$

where

$$N_C = CAM\ size\ (words)$$

$$C_p = CAM\ pollution\ mean$$

In the example, effective CAM size is:

$$Eff\ CAM\ Size = 8192\ (1 - .771) = 1876\ words$$

Analyzing the cumulative temporal locality characteristics of symbolic workloads, Hobart mapped the effective CAM

sizes to corresponding hit probabilities on old-old transitions (Hobart, 1989:103). Using this graph (Appendix C), the $W_C$ can be estimated for the effective CAM size of 1876 words:

$$W_C = 0.96$$

In order to calculate the probability of referencing a prefetched CAM block, $P_{CB}$, we must determine the number of old to new reference hits resulting from references to previously prefetched CAM blocks. First, the total number of additional references accessed within a CAM prefetched block, $A_R$, is calculated. From the CAM pollution ($C_p$) resulting from no prefetching (Table 4.3), (1 - $C_p$) or 43.9% of the demand-fetched references are rereferenced on average. We can note that if none of the prefetched words were referenced, the CAM pollution would be:

$$C_p = 1 - \frac{.439}{4} = 0.890$$

However, if all three of the prefetched references are, in fact, referenced, then the CAM pollution would be:

$$C_p = 1 - (\frac{.439}{4} + \frac{3}{4}) = 0.140$$

Our actual cache pollution must be between these two extremes. The following equation can be formed to determine the average number of additional references, $A_R$:

$$C_p = 1 - (\frac{(1 - C_p)}{B_C} + \frac{A_R}{B_C})$$

Solving for $A_R$ gives:

$$A_R = (B_C - 1)\ (1 - C_P)$$

Using this equation in our example, we have:

$$A_R = (3)\ (1 - .771) = 0.687$$

Therefore, 0.687 additional references are referenced in the CAM block on average.

Next, the number of references occurring within a CAM block during new to new referencing must be determined. Using a probability decision tree and summing all expected value outcomes, 1.41 references were found to be referenced during new to new referencing within a prefetched block of 4 words. Thus, only 0.41 out of the additional three references are referenced on average during new to new transitions. However, when the CAM spatially prefetches a block from main memory, all or a portion of the CAM block will also be prefetched to the SLC depending on where the requested address is located within the CAM block. Since the forward references (above the requested address) are prefetched to the SLC, new to new referencing within the CAM can only occur from those references within the CAM block which were not prefetched to the SLC (below the requested address). For example, if all four references in the CAM block are prefetched into the SLC, then no new to new references from that block can take place in the CAM. At the other extreme, if only the requested

address in the CAM block is prefetched to the SLC, then 0.41 new to new references from that block can take place in the CAM. The actual number of new to new references within the CAM block must be between these two extremes. Assuming one-half of the CAM block is prefetched into the SLC on average, we can also assume that only 0.2 additional references of the 3 prefetched references is referenced before transitioning back to the old reference state.

From the CAM pollution which occurs with no prefetching, we know that 0.439 references are accessed on average. By subtracting 0.439 references and the 0.2 new-new references from the total number of additional references (0.687), the number of old-new reference hits resulting from references to previously prefetched CAM blocks is 0.048 references.

Based on the state transition probability, $P_{ON}$ (.013), 13 out of every 1000 references can be expected to be old to new transitions. In this research, simulations revealed that if a prefetched address is to be referenced, it is first referenced within 256 memory accesses after being prefetched. Therefore, we can expect 3.328 (.013 x 256) to be referenced during the next 256 transitions.

The probability of referencing a previously prefetched CAM block, $P_{CB}$, can now be determined by dividing the number of old-new reference hits (0.048) by the expected number of old-new references during the "lifetime" of a prefetched reference (3.328) times the expected number of new-new

references (1.2) that will occur in the new reference state:

$$P_{CB} = \frac{0.048}{3.994} = 0.012$$

To calculate the probability of a CAM prefetch block reference during a new to new transition ($P_{CP}$), the expected value of number of new to new references, $U_{CP}$, can be set equal to the 0.2 additional new to new references. $U_{CP}$ is treated as a sum of geometric series using $P_{CP}$. From this, $P_{CP}$ can be calculated as follows:

$$U_{CP} = \frac{P_{CP}}{(1 - P_{CP})} = 0.2$$

Therefore

$$P_{CP} = \frac{.2}{1 + .2} = 0.167$$

Substituting the state transition probabilities, $W_C$, $P_{CB}$, and $P_{CP}$, the CAM hit probability equation can be solved:

$$P_{CAM} = \frac{.371\ (.96 + (.013)\ (.012)) + (.013)\ (.167)}{.384} = 0.932$$

In table 4.5, the CAM cache hit probability computations are compared with the measured mean values from the simulations. As shown, the predicted hit rates are slightly outside one standard deviation range from the measured hit rates. The calculated hit rates consistently overestimate the measured means. To account for this difference, future research may investigate how the effective CAM size may be further reduced

Table 4.5: CAM Cache Hit Probability Comparisons

SLC Size = 512 CAM Size = 8192
SLC Block (varied) CAM Block = 4

| SLC Block | 4 | 8 | 16 | 32 |
|---|---|---|---|---|
| Equation | .932 | .924 | .904 | .893 |
| Measured | | | | |
| Mean: | .855 | .827 | .802 | .777 |
| Std Dev: | .063 | .080 | .100 | .106 |

from spatial prefetching. The resulting smaller effective CAM size would decrease the calculated CAM hit probability.

## 4.4 Structural Locality Cache Hit Probability

Hobart developed a two-state Markov model to predict SLC hit rates when structural locality prefetching is used (Hobart, 1989:107-112). As shown in Figure 4.3, the model uses state transitions to represent the probabilities of various types of SLC referencing behavior. This model assumes no spatial locality prefetching.

A SLC hit can occur in two ways. Given a current old reference, a hit can take place if the next reference is also old and has not been replaced in the SLC. Since the SLC employs structural prefetching, the probability of this type of hit is calculated using the *SSD* and *NSSD* state transitions: $P_{SSD} + P_{NSSD}\, W_S$ where $P_{SSD}$ is the probability of an old to SSD state transition; $P_{NSSD}$ is the probability of an old to NSSD state transition; $W_S$ is effective cache size hit probability.

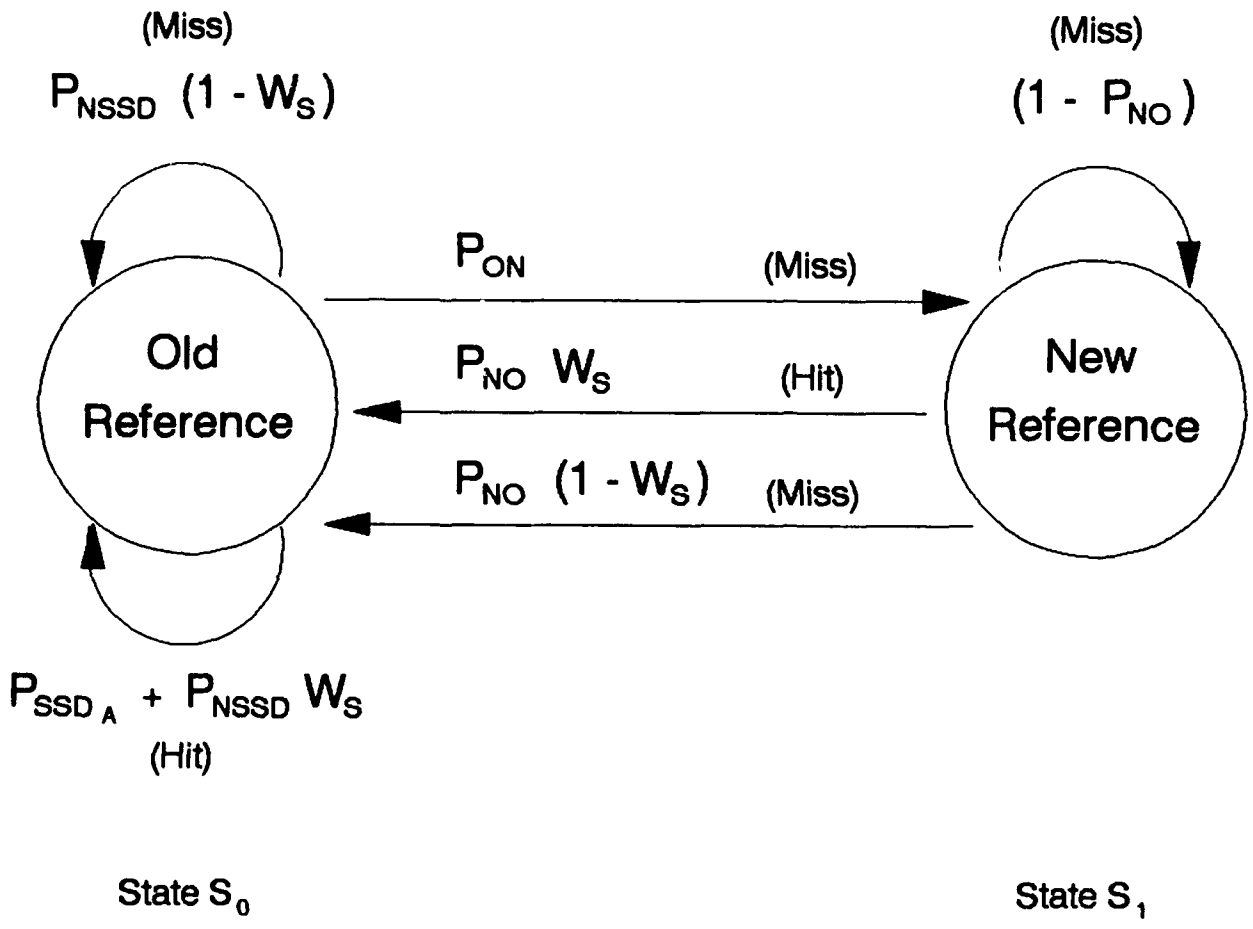



Figure 4.3: Markov Model for SLC Referencing with Structural Locality Prefetching but No Spatial Locality Prefetching

The other way a SLC hit can occur is during a new to old state transition: $P_{NO}\, W_S$. This probability is similar to the hit probability in the CAM model. Unlike the CAM model, a hit cannot occur on a new to new state transition since this model assumes that no spatial locality prefetching is being used in the CAM.

To incorporate the effects of spatial locality prefetching, my model adds two state transitions which can produce hits. As shown in Figure 4.4, these hit probabilities are similar to the corresponding probabilities in the modified CAM model (Figure 4.2). $P_{SB}$ is the probability of referencing spatially prefetched CAM data during an old to new transition. $P_{SP}$ is the probability of a reference to the same CAM block during a new to new reference.

Using the modified Markov model, the following equation is derived to determine the expected SLC hit rate, $P_{SLC}$:

$$P_{SLC} = ((P_{SSD_A} + P_{NSSD}\, W_S) + P_{SB} P_{ON}) \frac{P_{NO}}{P_{NO} + P_{ON}} + (P_{NO} W_S + P_{SP}) \frac{P_{ON}}{P_{NO} + P_{ON}}$$

which simplifies to

$$P_{SLC} = \frac{P_{NO}\ (P_{SSD_A} + (P_{NSSD} + P_{ON})\ W_S + P_{ON}\, P_{SB}) \ + \ P_{ON}\, P_{SP}}{P_{NO} + P_{ON}}$$

In order to determine $W_S$, the effective SLC size must be calculated as follows:

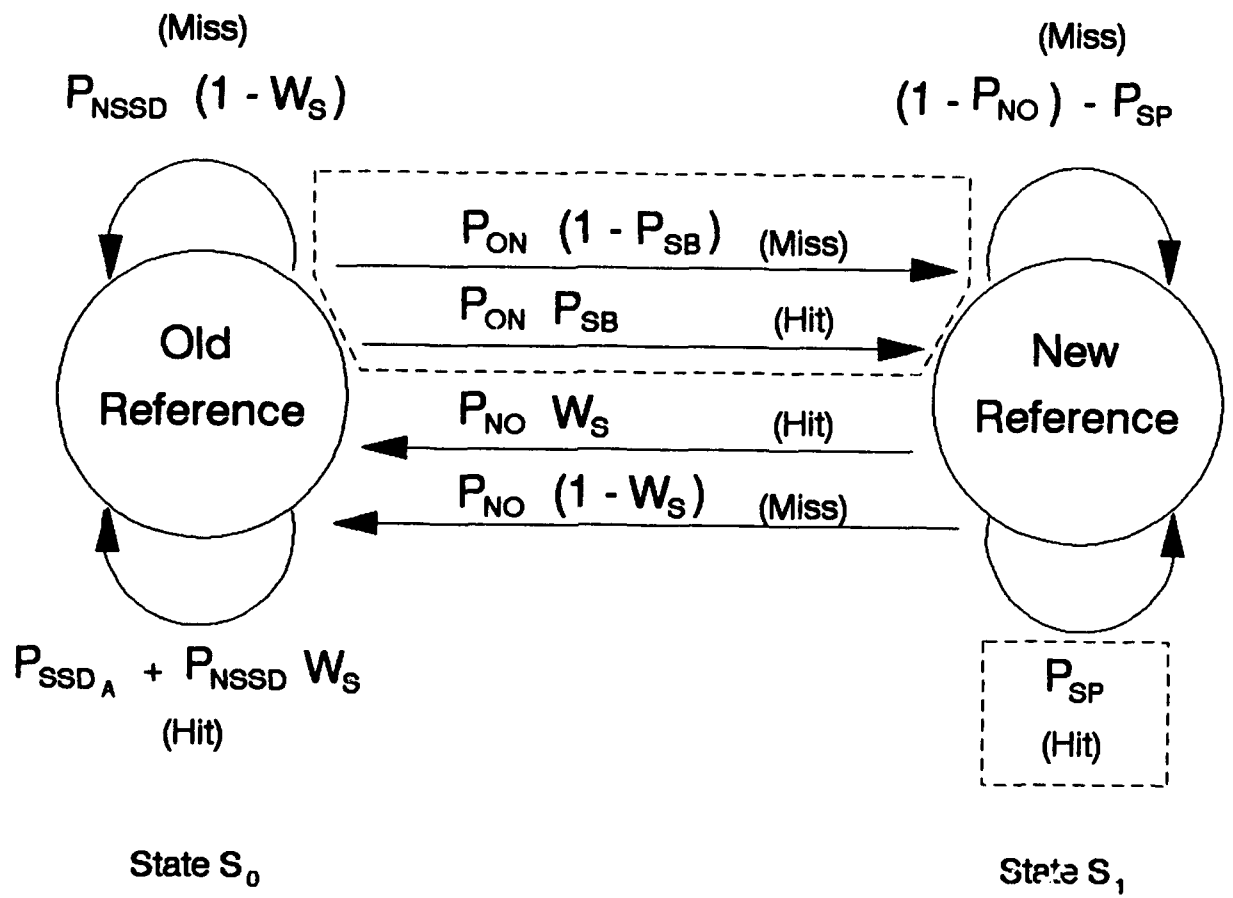



Figure 4.4: Modified Markov Model for SLC Referencing with Structural Locality Prefetching and Spatial Locality Prefetching

$$Eff\ SLC\ Size = N_S\ (E_{NN} + E_{SS})$$

where

$$N_S = SLC\ size\ (words)$$

$$E_{NN} = expected\ \%\ of\ new\text{-}new\ reference\ hits\ occurring\ within\ the\ most\ recently\ prefetched\ SLC\ block$$

$$E_{SS} = expected\ \%\ of\ old\text{-}new\ reference\ hits\ occurring\ within\ previously\ prefetched\ SLC\ blocks$$

Given the assumption that on average one-half of the CAM prefetched block (on a CAM miss) is prefetched into the SLC, the expected number of new-new references ($E_{NN}$) is 1.2 out of 4 references (0.3). Thus, only 0.2 of the three additional prefetched SLC references are accessed on average.

$E_{SS}$ is calculated as follows. First, the total number of additional references used within a SLC block, $A_R$, is determined (discussed in Section 4.3):

$$A_R = (B_S - 1)\ (1 - S_P)$$

where

$$B_S = SLC\ block\ size\ (words)$$

$$S_P = SLC\ pollution\ mean$$

Using the SLC block size of four words, we have:

$$A_R = 3\ (.392) = 1.176$$

From the SLC pollution occurring with no prefetching, we know that 0.511 demand-fetched references are rereferenced on

average. Subtracting the 0.511 references and the 0.2 additional new-new references from the total number of additional references (1.176), the number of additional old-new reference hits resulting from references to previously prefetched SLC blocks is 0.465. From this, $E_{SS}$ can be determined:

$$E_{SS} = \frac{.465}{4} = .116$$

We can now solve for the effective CAM size:

$$Eff\ SLC\ Size = 512\ (.3 + .116) = 213\ words$$

$W_S$ can now be estimated (Appendix C) for 213 words:

$$W_S = 0.67$$

$P_{SB}$ is calculated in the same manner as $P_{CB}$ is calculated for the CAM cache. It is determined by dividing the number of old-new reference hits (.465) by the expected number of old-new references during the "lifetime" of a prefetched reference (3.328) times the expected number of new-new references (1.2) that will occur in the new reference state:

$$P_{SB} = \frac{.465}{3.994} = 0.116$$

Since SLC blocks prefetch CAM blocks, $P_{SB}$ remains constant for all SLC block sizes.

Given consecutive new-new references only occur within a given block, $P_{SP}$ is calculated in a manner similar to $P_{CP}$:

$$P_{SP} = \frac{.2}{(1 + .2)} = 0.167$$

Substituting the state transition probabilities, $W_S$, $P_{SB}$, and $P_{SP}$, the SLC hit probability equation can be solved:

$$P_{SLC} = \frac{.371\,(.48 + (.449)\,.67 + (.013)\,.116) + (.013)\,.167}{.384} = 0.761$$

In table 4.6, the SLC hit probability computations are compared with the measured mean values from the simulations. As shown, the predicted hit rates are relatively close to the measured means. Hit rates for the SLC block sizes of 8 and 16 words fall within one standard deviation of the measured means. Hit rates for the SLC block sizes of 4 and 32 words are slightly outside one standard deviation range. The calculated rates consistently underestimate the measured rates. Although the overwhelming majority of SLC prefetches take place from the CAM (allowing for a full SLC block), the portion of SLC prefetches initiated from main memory may still have an impact on pollution. Reduced pollution can increase the effective size of SLC. And, in turn, increase the calculated SLC hit probability.

Table 4.6: SLC Hit Probability Comparisons

SLC Size = 512 CAM Size = 8192
SLC Block (varied) CAM Block = 4

| SLC Block | 4 | 8 | 16 | 32 |
|---|---|---|---|---|
| Equation | .761 | .836 | .846 | .839 |
| Measured | | | | |
| Mean: | .859 | .880 | .893 | .908 |
| Std Dev: | .061 | .055 | .051 | .044 |

In addition, as in the case of the CAM cache, future research may also investigate the possibility that additional SLC hits may occur as a result of new-new references which take place within multiple blocks. These additional hits would be represented by another new-new state transition. The resulting SLC hits would increase the calculated SLC hit probability.

## 4.5 Performance Analysis

The effective memory access times were measured for the four sets of SLC and CAM cache block parameters (Table 4.3). As presented in Chapter 3, the effective memory access time, $t_a$, is calculated as follows:

$$t_a = t_{SLC}P_{SLC} + t_{CAM}(1 - P_{SLC})\,P_{CAM} + t_{MEM}(1 - P_{CAM})\,(1 - P_{SLC})$$

where the cycle times for the SLC, CAM, and main memory are 1, 4, and 32 clock cycles, respectively.

To serve as a baseline comparison, the effective memory access time is used for the same SLC and CAM cache sizes when no prefetching is involved. From Table 4.3, the measured mean effective memory access time for no prefetching is 2.822. Calling this the baseline access time *Tb*, the performance speedup *S* due to prefetching is defined as (Hobart, 1989:113):

$$S = \left(\frac{t_b}{t_n} - 1\right) 100 \qquad (percent)$$

As shown in Table 4.7, the speedup due to spatial and

structural prefetching occurred for all SLC block sizes. For this set of parameters, increasing the SLC block size results in performance improvement. As the SLC block size nears the SLC size, performance gains would be expected to decrease. In this set, the largest SLC block of 32 words did not adversely impact the SLC hit ratio since a sufficient number of blocks (16) could still be stored in the SLC. As a result, it produced the smallest effective memory access time.

Table 4.7: Speedup Due to Spatial and Structural Prefetching

SLC Size = 512 CAM Size = 8192
No Prefetch *Tb* = 5.25

| CAM Prefetch Block = 4 | |
|---|---|
| SLC Block | *S* (%) |
| 4 | 6.5 |
| 8 | 8.2 |
| 16 | 9.6 |
| 32 | 12.4 |

## 4.6 Summary

This chapter has shown how Hobart's memory referencing models were modified to incorporate the effects of spatial prefetching in the CAM and structural prefetching in the SLC. CAM and SLC hit probability equations were derived from these models. Using the measurements from the trace-driven simulations, the expected hit rates were calculated for both caches using different sets of block size parameters. In

particular, cache pollution estimates were used to analyze the effects of spatial prefetching on structural locality.

# Chapter 5

# Conclusion and Recommendations

## 5.1 Main Contributions

The principal contribution of this research is characterizing the effects of spatial prefetching on structural locality in the proposed memory subsystem. This research shows that performance gains through structural locality prefetching are still possible even when spatial locality prefetching is being used in the lower level cache.

Existing memory referencing models are modified to incorporate the combined use of structural locality and spatial locality prefetching. From these models, equations are derived to predict the expected hit rates of the SLC and CAM caches. Targeting a representative sample of symbolic workloads, trace-driven simulations provide performance measurements for different combinations of SLC and CAM cache block sizes. Combined with the state transition probabilities of the modified Markov memory referencing models, these measurements are used to solve the hit probability equations.

The purpose of this research is not to exhaustively measure performance for all combinations of cache parameters. Instead, it provides an extended memory referencing model to analytically predict the hit rates and the effective memory

access times for a range of SLC block sizes.

The experimental methodology for this research is also a significant contribution. The cache simulator provides a reliable tool for not only meeting the demands of this research, but also future research as well. Built using the Ada language, the simulator is designed to facilitate tailoring it to meet new research requirements.

## 8.2 Additional Applications of This Research

Several approaches exist for continuing this research. First, the modified memory referencing models should be applied to other types of workloads such as numeric and data-processing. Expanding the applicability to various workload types will increase the usefulness of the models. In addition, the effects of varying other parameters besides SLC block size should be examined.

Another research application is to further analyze the temporal behavior within the SLC and CAM cache. Specifically, it would be useful to differentiate between prefetched instructions and data. Further investigation is also required to characterize this temporal behavior over a wide range of cache parameters. The cache simulator has already been designed to provide data for this analysis. As described in Chapter 3, the *Generate_Ref_ Frequency_Package* outputs a reference file containing temporal reference information for both caches.

Another research course would be to study the impact of combined structural locality and spatial locality prefetching on individual reference types: instruction fetches, data reads, and data writes.

# Appendix A: Cache Simulator Source Code

```
package  LinkedLists_Package is

   type LinkedListNode;
   type NodePointer is access LinkedListNode;

   type LinkedListNode is
      record
         Addr    : integer;
         NumRef  : integer := 0;
         Next    : NodePointer := null;
      end record;

   type List is
      record
         Next:  NodePointer := null;
         Tail:  NodePointer := null;
      end record;

-- create a node (address to be stored in cache)
   function MakeNode (Address: integer) return NodePointer;

-- find node containing matching address with P1
   function Search (L: List; Address: integer)
                    return NodePointer;

-- add node P1 to front of list
   procedure AddToFront (L: in out List; P1: NodePointer);

-- add node P1 to rear of list
   procedure AddToRear (L: in out List; P1: NodePointer);

-- insert node P1 in order by number of passed references
--    before being first referenced
procedure Insert_In_Order (L: in out List; P1: NodePointer);

-- delete node P1 from list
   procedure Delete (L: in out List; P1: NodePointer);

end LinkedLists_Package;
```

```
with Unchecked_Deallocation;

package body  LinkedLists_Package  is

   procedure Deallocate_Node is new Unchecked_Deallocation
             (Object => LinkedListNode,
              Name   => NodePointer);

   function MakeNode (Address: integer)
                      return NodePointer is
      p: NodePointer;
   begin
      p := new LinkedListNode;
      p.Addr := Address;
      return p;
   end MakeNode;

   function Search (L: List; Address: integer)
                    return NodePointer is
      p: NodePointer := L.Next;
   begin
      while p /= null and then p.Addr /= Address loop
         p := p.Next;
      end loop;
      return p;
   end Search;

   procedure AddToFront (L: in out List; P1: NodePointer) is
   begin
      P1.Next := L.Next;
      L.Next := P1;
      if L.Tail = null then L.Tail := P1; end if;
   end AddToFront;

   procedure AddToRear (L: in out List; P1: NodePointer) is
   begin
      if L.Next = null then
         AddToFront (L, P1);
      end if;
      L.Tail.Next := P1;
      L.Tail := P1;
   end AddToRear;

   procedure Insert_In_Order (L: in out List; P1: NodePointer) is
      p: NodePointer := L.Next;
      q: NodePointer;
   begin
      if p = null then
         AddToFront (L, P1);
      elsif p.NumRef > P1.NumRef then
         AddToFront (L, P1);
      else
         while p /= null and then
            p.NumRef < P1.NumRef + 1 loop
            q := p;
            p := p.Next;
         end loop;
         if p = null then
            AddToRear (L, P1);
         else
            q.Next := P1;
            P1.Next := p;
         end if;
      end if;
   end Insert_In_Order;
```

```
   procedure Delete (L: in out List; P1: NodePointer) is
      p: NodePointer := L.Next;
      q: NodePointer := null;
   begin
      if P1 = p then L.Next := P1.Next;
      else
         while p /= null and then p /= P1 loop
            q := p;
            p := p.Next;
         end loop;
         q.Next := P1.Next;
         Deallocate_Node (p);
      end if;
      if L.Tail = P1 then L.Tail := q; end if;
   end Delete;

end LinkedLists_Package;
```

```
generic

   type Object is limited private;
   type Name is access Object;

procedure Unchecked_Deallocation (x: in out Name);
```

```
package  CircularQ_Package is

   type index is range 1 .. 32768;
   type Array_Type is array (index) of integer;
   type Array_Ptr_Type is access Array_Type;

type Queue is
      record
         Address      : Array_Ptr_Type := new Array_Type;
         Ref_Count    : Array_Ptr_Type := new Array_Type;
         head         : index;
         tail         : index;
      end record;

   procedure Enqueue (Q: in out Queue;
                      CAM_Size_Index: in index;
                      Reference: in integer);

   procedure Dequeue (Q: in out Queue;
                      CAM_Size_Index: in index);

   procedure SearchQ (Q: in Queue;
                      Reference: in integer;
                      CAM_Size_Index: in index;
                      Position: in out index;
                      Found: in out boolean);

end CircularQ_Package;
```

```
package body CircularQ_Package is

   procedure Enqueue (Q: in out Queue;
                      CAM_Size_Index: in index;
                      Reference: in integer) is

   begin
      Q.tail := (Q.tail mod CAM_Size_Index) + 1;
      Q.Address (Q.tail) := Reference;
      Q.Ref_Count (Q.tail) := 0;

   end Enqueue;

   procedure Dequeue (Q: in out Queue;
                      CAM_Size_Index: in index) is

   begin
      Q.head := (Q.head mod CAM_Size_Index) + 1;

   end Dequeue;

   procedure SearchQ (Q: in Queue;
                      Reference: in integer;
                      CAM_Size_Index: in index;
                      Position: in out index;
                      Found: in out boolean) is

   begin
      for i in 1 .. CAM_Size_Index loop
         Position := (Position mod CAM_Size_Index) + 1;
         if Q.Address (Position) = Reference
            then Found := true;
            exit;
         end if;
      end loop;

   end SearchQ;

end CircularQ_Package;
```

```
package Addr_Record_Package is

   type Addr_Record is
      record
         The_Type : character;
         Address  : integer;
      end record;

end Addr_Record_Package;
```

```
ith Text_IO, Addr_Record_Package,
    LinkedLists_Package, CircularQ_Package,
    Fetch_Address_Package, Determine_Type_Package,
    Serv_Instr_Fetch_Package, Serv_Data_Read_Package,
    Serv_Data_Write_Package, Compute_Miss_Ratios_Package,
    Compute_Memory_Access_Time_Package,
    Compute_Cache_Pollution_Package,
    Generate_Ref_Frequency_List_Package;
se  Text_IO, Addr_Record_Package,
    LinkedLists_Package, CircularQ_Package,
    Fetch_Address_Package, Determine_Type_Package,
    Serv_Instr_Fetch_Package, Serv_Data_Read_Package,
    Serv_Data_Write_Package, Compute_Miss_Ratios_Package,
    Compute_Memory_Access_Time_Package,
    Compute_Cache_Pollution_Package,
    Generate_Ref_Frequency_List_Package;

  procedure Cache_Simulator is

    package Type_Integer_IO is new integer_IO (integer);
    use Type_Integer_IO;
    package Index_Integer_IO is new integer_IO (index);
    use Index_Integer_IO;

    Input_File          : File_Type;
    Output_File         : File_Type;
    Reference_File      : File_Type;
    Memory_Ref          : Addr_Record;
    Num_Ref             : natural := 0;
    Total_Num_Ref       : natural := 0;
    Type_Ref            : character;
    SLC                 : List;
    CAM                 : Queue;
    SLC_Ref_List        : List;
    CAM_Ref_List        : List;
    SLC_Miss            : natural := 0;
    CAM_Miss            : natural := 0;
    SLC_Total_Refs      : natural := 0;
    CAM_Total_Refs      : natural := 0;
    Temp_SLC_Size       : natural := 0;
    Temp_CAM_Size       : natural := 0;
    SLC_Non_Ref         : natural := 0;
    CAM_Non_Ref         : natural := 0;
    SLC_Total_Prefetch  : natural := 0;
    CAM_Total_Prefetch  : natural := 0;
    SLC_MR, CAM_MR      : float;
    CAM_Size_Index      : index;
    In_String           : string (1..15);
    In_Length           : natural;
    New_Space           : natural := 1;
    Last_Space          : natural := 0;

-- ************************************************************
-- *                  CACHE   PARAMETERS                      *
-- ************************************************************

    SLC_Size            : natural;
    CAM_Size            : natural;
    SLC_Line_Size       : natural;
    CAM_Line_Size       : natural;

-- ************************************************************
```



```
    begin
       new_line;
       put ("Please enter the following:"); new_line(2);
```

```
put ("Statistics Filename:  ");
get_line (In_String, In_Length);
while ((New_Space < In_Length) and
   (In_String (New_Space .. New_Space) /= " ")) loop
   New_Space := New_Space + 1;
end loop;
create (Output_File, Out_File,
        In_String ((Last_Space + 1) .. New_Space));
New_Space := 1; Last_Space := 0;

put ("Reference Filename:  ");
get_line (In_String, In_Length);
while ((New_Space < In_Length) and
   (In_String (New_Space .. New_Space) /= " ")) loop
   New_Space := New_Space + 1;
end loop;
create (Reference_File, Out_File,
        In_String ((Last_Space + 1) .. New_Space));
New_Space := 1; Last_Space := 0;

put ("Trace Filename:  ");
get_line (In_String, In_Length);
while ((New_Space < In_Length) and
   (In_String (New_Space .. New_Space) /= " ")) loop
   New_Space := New_Space + 1;
end loop;
open (Input_File, In_File,
      In_String ((Last_Space + 1) .. New_Space));
new_line;

put ("*** Cache Parameters ***"); new_line;
put ("SLC Size:  "); get (SLC_Size);
put ("SLC Line Size:  "); get (SLC_Line_Size);
put ("CAM Size:  "); get (CAM_Size);
put ("CAM Line Size:  "); get (CAM_Line_Size); new_line;

-------------------------------------------------
--  Heading info:  stats & reference files.
-------------------------------------------------
put (Output_File, "Address Trace:  ");
put (Output_File, In_String((Last_Space+1)..New_Space));
put (Reference_File, "Address Trace:  ");
put (Reference_File, In_String((Last_Space+1)..New_Space));
set_line (Output_File, 2);
set_line (Reference_File, 2);
put (Output_File, "SLC Size:  ");
put (Output_File, SLC_Size);
put (Output_File, "          SLC Line:  ");
put (Output_File, SLC_Line_Size);
put (Reference_File, "SLC Size:  ");
put (Reference_File, SLC_Size);
put (Reference_File, "          SLC Line:  ");
put (Reference_File, SLC_Line_Size);
set_line (Output_File, 3);
set_line (Reference_File, 3);
put (Output_File, "CAM Size:  ");
put (Output_File, CAM_Size);
put (Output_File, "          CAM Line:  ");
put (Output_File, CAM_Line_Size);
put (Reference_File, "CAM Size:  ");
put (Reference_File, CAM_Size);
put (Reference_File, "          CAM Line:  ");
put (Reference_File, CAM_Line_Size);
set_line (Output_File, 5);
set_line (Reference_File, 5);
```

```
-------------------------------------------------
-- Initialize CAM head & tail to CAM size.
-------------------------------------------------
CAM_Size_Index := index (CAM_Size);
CAM.head := CAM_Size_Index;
CAM.tail := CAM_Size_Index;
-------------------------------------------------

while not End_Of_File (Input_File) loop

   Load_Record (Input_File, Memory_Ref);
   Type_Ref := Address_Type (Memory_Ref);

   loop
      case Type_Ref is

         when '0' =>
            Num_Ref := Num_Ref + 1;
            Serv_Data_Read (Input_File, Memory_Ref,
                  SLC, CAM, SLC_Ref_List, CAM_Ref_List,
                  SLC_Miss, CAM_Miss,
                  SLC_Total_Refs, CAM_Total_Refs,
                  SLC_Non_Ref, CAM_Non_Ref,
                  SLC_Total_Prefetch, CAM_Total_Prefetch,
                  SLC_Size, CAM_Size, SLC_Line_Size,
                  CAM_Line_Size, Temp_SLC_Size,
                  Temp_CAM_Size); exit;

         when '1' =>
            Num_Ref := Num_Ref + 1;
            Serv_Data_Write (Input_File, Memory_Ref,
                  SLC, CAM, SLC_Ref_List, CAM_Ref_List,
                  SLC_Miss, CAM_Miss,
                  SLC_Total_Refs, CAM_Total_Refs,
                  SLC_Non_Ref, CAM_Non_Ref,
                  SLC_Total_Prefetch, CAM_Total_Prefetch,
                  SLC_Size, CAM_Size, SLC_Line_Size,
                  CAM_Line_Size, Temp_SLC_Size,
                  Temp_CAM_Size); exit;

         when '2' =>
            Num_Ref := Num_Ref + 1;
            Serv_Instr_Fetch (Input_File, Memory_Ref,
                  SLC, CAM, SLC_Ref_List, CAM_Ref_List,
                  SLC_Miss, CAM_Miss,
                  SLC_Total_Refs, CAM_Total_Refs,
                  SLC_Non_Ref, CAM_Non_Ref,
                  SLC_Total_Prefetch, CAM_Total_Prefetch,
                  SLC_Size, CAM_Size, SLC_Line_Size,
                  CAM_Line_Size, Temp_SLC_Size,
                  Temp_CAM_Size); exit;

         when others => exit;

      end case;
   end loop;

   if Num_Ref = 20000 then
      Total_Num_Ref := Total_Num_Ref + Num_Ref;

      ------------------------------------------------
      -- Compute SLC & CAM miss ratios
      --    for every 20000 memory references
      ------------------------------------------------
      Compute_Miss_Ratios (SLC_Miss, CAM_Miss,
            SLC_Total_Refs, CAM_Total_Refs,
```

```
                Total_Num_Ref, SLC_MR,
                CAM_MR, Output_File);
            Num_Ref := 0;

        elsif End_Of_File (Input_File) then
            Total_Num_Ref := Total_Num_Ref + Num_Ref;

            ------------------------------------------
            -- Compute final SLC & CAM miss ratios
            ------------------------------------------
            Compute_Miss_Ratios (SLC_Miss, CAM_Miss,
                SLC_Total_Refs, CAM_Total_Refs,
                Total_Num_Ref, SLC_MR,
                CAM_MR, Output_File);

            ------------------------------------------
            -- Compute average memory access time
            ------------------------------------------
            Compute_Memory_Access_Time (SLC_MR, CAM_MR,
                                        Output_File);

            ------------------------------------------
            -- Compute SLC & CAM pollution
            ------------------------------------------
            Compute_Cache_Pollution (SLC, CAM, SLC_Non_Ref,
               CAM_Non_Ref, SLC_Total_Prefetch, CAM_Total_Prefetch,
               Temp_CAM_Size, Output_File);

            ------------------------------------------
            -- Generate reference frequency list
            ------------------------------------------
            Generate_Ref_Frequency_List (SLC_Ref_List,
               CAM_Ref_List, SLC_Non_Ref, CAM_Non_Ref,
               Reference_File);
            ------------------------------------------

            new_line (2);
            put_line ("*****  End of Simulation  *****");
            exit;
        end if;

    end loop;

  close (Input_File);
  close (Output_File);
  close (Reference_File);

end Cache_Simulator;
```

```
with Addr_Record_Package,
     LinkedLists_Package, CircularQ_Package, Text_IO;
use  Addr_Record_Package,
     LinkedLists_Package, CircularQ_Package, Text_IO;

package Serv_Data_Read_Package is

  procedure Serv_Data_Read
           (Input_File           : in File_Type;
            Memory_Ref           : in Addr_Record;
            SLC                  : in out List;
            CAM                  : in out Queue;
            SLC_Ref_List         : in out List;
            CAM_Ref_List         : in out List;
            SLC_Miss             : in out natural;
            CAM_Miss             : in out natural;
            SLC_Total_Refs       : in out natural;
            CAM_Total_Refs       : in out natural;
            SLC_Non_Ref          : in out natural;
            CAM_Non_Ref          : in out natural;
            SLC_Total_Prefetch   : in out natural;
            CAM_Total_Prefetch   : in out natural;
            SLC_Size             : in natural;
            CAM_Size             : in natural;
            SLC_Line_Size        : in natural;
            CAM_Line_Size        : in natural;
            Temp_SLC_Size        : in out natural;
            Temp_CAM_Size        : in out natural);

end Serv_Data_Read_Package;
```

```
package body Serv_Data_Read_Package is

   procedure Serv_Data_Read
             (Input_File          : in File_Type;
              Memory_Ref          : in Addr_Record;
              SLC                 : in out List;
              CAM                 : in out Queue;
              SLC_Ref_List        : in out List;
              CAM_Ref_List        : in out List;
              SLC_Miss            : in out natural;
              CAM_Miss            : in out natural;
              SLC_Total_Refs      : in out natural;
              CAM_Total_Refs      : in out natural;
              SLC_Non_Ref         : in out natural;
              CAM_Non_Ref         : in out natural;
              SLC_Total_Prefetch  : in out natural;
              CAM_Total_Prefetch  : in out natural;
              SLC_Size            : in natural;
              CAM_Size            : in natural;
              SLC_Line_Size       : in natural;
              CAM_Line_Size       : in natural;
              Temp_SLC_Size       : in out natural;
              Temp_CAM_Size       : in out natural) is

      package Index_Integer_IO is new integer_IO (index);
      use Index_Integer_IO;

      Reference            : Addr_Record;
      Temp_Nd1, Temp_Nd2   : NodePointer;
      Temp_Nd3             : NodePointer;
      Position             : index := CAM.head;
      Found                : boolean := false;
      k                    : natural := 1;
      CAM_Prefetch_Addr    : integer;
      n,o,p                : integer;
      CAM_Size_Index, j    : index;
      Temp_CAM_Tail        : index;

   begin
   -- ****************************************
   -- Search for memory reference in the SLC.
   -- ****************************************

      SLC_Total_Refs := SLC_Total_Refs + 1;

      Temp_Nd1 := SLC.Next;
      while Temp_Nd1 /= null loop
         if Temp_Nd1.NumRef /= -1 then
            Temp_Nd1.NumRef := Temp_Nd1.NumRef + 1;
         end if;
         Temp_Nd1 := Temp_Nd1.Next;
      end loop;

      n := Memory_Ref.Address;
      Temp_Nd1 := MakeNode (Memory_Ref.Address);

      Temp_Nd2 := Search (SLC, n);

   -- ****************************************
   -- If the address is found in the SLC,
   -- then delete the address in the cache
   -- (linked list) and add it to the front
   -- of the list (most recently used).
   -- ****************************************

      if Temp_Nd2 /= null then
```

```
   AddToFront (SLC, Temp_Nd1);
   SLC.Next.NumRef := -1;
   if Temp_Nd2.NumRef /= -1 then
      Temp_Nd3 := MakeNode (Temp_Nd2.Addr);
      Temp_Nd3.NumRef := Temp_Nd2.NumRef;
      Insert_In_Order (SLC_Ref_List, Temp_Nd3);
   end if;
   Delete (SLC, Temp_Nd2);

else

-- ****************************************
-- Since a miss has occurred in the SLC,
-- search for the address in the CAM.
-- ****************************************

   SLC_Miss := SLC_Miss + 1;
   CAM_Total_Refs := CAM_Total_Refs + 1;
   CAM_Size_Index := index (CAM_Size);

   if Temp_CAM_Size /= 0 then
      for i in 1 .. Temp_CAM_Size loop
         j := index (i);
         if CAM.Ref_Count (j) /= -1 then
            CAM.Ref_Count (j) := CAM.Ref_Count (j) + 1;
         end if;
      end loop;
   end if;

   SearchQ (CAM, n, CAM_Size_Index, Position, Found);

-- ****************************************
-- If the address is found in the CAM, then
-- prefetch a line from the CAM into the SLC.
-- The SLC line is comprised of the requested
-- address + the addresses located after the
-- requested address (total equal to the SLC
-- line size).  This action represents a
-- spatial prefetch of the structural
-- locality captured in the CAM.
-- ****************************************

   if Found = true then          -- address found in CAM

      if CAM.Ref_Count (Position) /= -1 then
         Temp_Nd1 := MakeNode (CAM.Address (Position));
         Temp_Nd1.NumRef := CAM.Ref_Count (Position);
         Insert_In_Order (CAM_Ref_List, Temp_Nd1);
         CAM.Ref_Count (Position) := -1;
      end if;
      while Position /= CAM.tail
         and then k /= SLC_Line_Size loop
            k := k + 1;
            Position := Position mod CAM_Size_Index + 1;
         end loop;

      for i in 1 .. k loop

      -- ****************************************
      -- If the SLC is full, then delete the
      -- address located at the rear of the list
      -- (LRU replacement).
      -- ****************************************

         if Temp_SLC_Size = SLC_Size then
```

```
            if SLC.tail.NumRef /= -1 then
               SLC_Non_Ref := SLC_Non_Ref + 1;
            end if;

            Delete (SLC, SLC.tail);
            Temp_SLC_Size := Temp_SLC_Size - 1;

         end if;

      -- ***************************************

         Temp_Ndl := MakeNode (CAM.Address (Position));
         AddToFront (SLC, Temp_Ndl);
         SLC_Total_Prefetch := SLC_Total_Prefetch + 1;
         Temp_SLC_Size := Temp_SLC_Size + 1;
         if Position = 1 then
            Position := CAM_Size_Index;
         else
            Position := Position - 1;
         end if;

      end loop;

   else                              -- address not found in CAM

   -- ********************************************
   -- If a miss occurs in the CAM, then prefetch a
   -- line from main memory into the CAM.  The line
   -- will be comprised of the block of memory (equal
   -- to the CAM line size) in which the requested
   -- address is located.
   -- ********************************************

      CAM_Miss := CAM_Miss + 1;

   -- ********************************************
   -- If the CAM is full, then the delete the
   -- addresses (amount equal to the CAM line size)
   -- in the front of the CAM (FIFO replacement).
   -- ********************************************

      if Temp_CAM_Size = CAM_Size then
         for i in 1 .. CAM_Line_Size loop

            if CAM.Ref_Count (CAM.head mod CAM_Size_Index + 1)
               /= -1 then
               CAM_Non_Ref := CAM_Non_Ref + 1;
            end if;

            Dequeue (CAM, CAM_Size_Index);
            Temp_CAM_Size := Temp_CAM_Size - 1;

         end loop;
      end if;

   -- ********************************************

      o := n / CAM_Line_Size;
      CAM_Prefetch_Addr := o * CAM_Line_Size;

   -- ********************************************
   -- CAM prefetch for a positive integer address.
   -- ********************************************

      if n >= 0 then
         CAM_Prefetch_Addr := CAM_Prefetch_Addr - 1;
```

```
      for i in 1 .. CAM_Line_Size loop
         CAM_Prefetch_Addr := CAM_Prefetch_Addr + 1;
         Enqueue (CAM, CAM_Size_Index, CAM_Prefetch_Addr);
         CAM_Total_Prefetch := CAM_Total_Prefetch + 1;
         Temp_CAM_Size := Temp_CAM_Size + 1;
      end loop;
   else

   -- ********************************************
   -- CAM prefetch for a negative integer address.
   -- ********************************************

      if n rem CAM_Line_Size /= 0 then
         CAM_Prefetch_Addr := CAM_Prefetch_Addr - CAM_Line_Size;
      end if;
      for i in 1 .. CAM_Line_Size loop
         Enqueue (CAM, CAM_Size_Index, CAM_Prefetch_Addr);
         CAM_Total_Prefetch := CAM_Total_Prefetch + 1;
         CAM_Prefetch_Addr := CAM_Prefetch_Addr + 1;
         Temp_CAM_Size := Temp_CAM_Size + 1;
      end loop;
   end if;

-- ********************************************
-- Prefetch the addresses from the CAM to the SLC
-- starting with the requested address and ending
-- with the last (tail of the circular queue)
-- address in the CAM.
-- ********************************************

   Temp_CAM_Tail := CAM.tail;

   while CAM.Address (Temp_CAM_Tail) /= n loop
      Temp_Nd1 := MakeNode (CAM.Address (Temp_CAM_Tail));

      if Temp_SLC_Size = SLC_Size then

         if SLC.tail.NumRef /= -1 then
            SLC_Non_Ref := SLC_Non_Ref + 1;
         end if;

         Delete (SLC, SLC.tail);
         Temp_SLC_Size := Temp_SLC_Size - 1;

      end if;

      AddToFront (SLC, Temp_Nd1);
      SLC_Total_Prefetch := SLC_Total_Prefetch + 1;
      Temp_SLC_Size := Temp_SLC_Size + 1;
      if Temp_CAM_Tail = 1 then
         Temp_CAM_Tail := CAM_Size_Index;
      else
         Temp_CAM_Tail := Temp_CAM_Tail - 1;
      end if;

   end loop;

   if Temp_SLC_Size = SLC_Size then

      if SLC.tail.NumRef /= -1 then
         SLC_Non_Ref := SLC_Non_Ref + 1;
      end if;
```



```
      Delete (SLC, SLC.tail);
      Temp_SLC_Size := Temp_SLC_Size - 1;
```

```
            end if;

            Temp_Nd1 := MakeNode (CAM.Address (Temp_CAM_Tail));
            AddToFront (SLC, Temp_Nd1);
            SLC_Total_Prefetch := SLC_Total_Prefetch + 1;
            Temp_SLC_Size := Temp_SLC_Size + 1;

         end if;

      end if;

   end Serv_Data_Read;
end Serv_Data_Read_Package;
```

```
with Addr_Record_Package,
     LinkedLists_Package, CircularQ_Package, Text_IO;
use  Addr_Record_Package,
     LinkedLists_Package, CircularQ_Package, Text_IO;

package Serv_Data_Write_Package is

  procedure Serv_Data_Write
           (Input_File          : in File_Type;
            Memory_Ref          : in Addr_Record;
            SLC                 : in out List;
            CAM                 : in out Queue;
            SLC_Ref_List        : in out List;
            CAM_Ref_List        : in out List;
            SLC_Miss            : in out natural;
            CAM_Miss            : in out natural;
            SLC_Total_Refs      : in out natural;
            CAM_Total_Refs      : in out natural;
            SLC_Non_Ref         : in out natural;
            CAM_Non_Ref         : in out natural;
            SLC_Total_Prefetch  : in out natural;
            CAM_Total_Prefetch  : in out natural;
            SLC_Size            : in natural;
            CAM_Size            : in natural;
            SLC_Line_Size       : in natural;
            CAM_Line_Size       : in natural;
            Temp_SLC_Size       : in out natural;
            Temp_CAM_Size       : in out natural);

end Serv_Data_Write_Package;
```

```
package body Serv_Data_Write_Package is

   procedure Serv_Data_Write
            (Input_File          : in File_Type;
             Memory_Ref          : in Addr_Record;
             SLC                 : in out List;
             CAM                 : in out Queue;
             SLC_Ref_List        : in out List;
             CAM_Ref_List        : in out List;
             SLC_Miss            : in out natural;
             CAM_Miss            : in out natural;
             SLC_Total_Refs      : in out natural;
             CAM_Total_Refs      : in out natural;
             SLC_Non_Ref         : in out natural;
             CAM_Non_Ref         : in out natural;
             SLC_Total_Prefetch  : in out natural;
             CAM_Total_Prefetch  : in out natural;
             SLC_Size            : in natural;
             CAM_Size            : in natural;
             SLC_Line_Size       : in natural;
             CAM_Line_Size       : in natural;
             Temp_SLC_Size       : in out natural;
             Temp_CAM_Size       : in out natural) is

      package Index_Integer_IO is new integer_IO (index);
      use Index_Integer_IO;

      Reference            : Addr_Record;
      Temp_Nd1, Temp_Nd2   : NodePointer;
      Temp_Nd3             : NodePointer;
      Position             : index := CAM.head;
      Found                : boolean := false;
      k                    : natural := 1;
      CAM_Prefetch_Addr    : integer;
      n,o,p                : integer;
      CAM_Size_Index, j    : index;
      Temp_CAM_Tail        : index;

   begin
   -- ****************************************
   -- Search for memory reference in the SLC.
   -- ****************************************

      SLC_Total_Refs := SLC_Total_Refs + 1;

      Temp_Nd1 := SLC.Next;
      while Temp_Nd1 /= null loop
         if Temp_Nd1.NumRef /= -1 then
            Temp_Nd1.NumRef := Temp_Nd1.NumRef + 1;
         end if;
         Temp_Nd1 := Temp_Nd1.Next;
      end loop;

      n := Memory_Ref.Address;
      Temp_Nd1 := MakeNode (Memory_Ref.Address);

      Temp_Nd2 := Search (SLC, n);

   -- ****************************************
   -- If the address is found in the SLC,
   -- then delete the address in the cache
   -- (linked list) and add it to the front
   -- of the list (most recently used).
   -- ****************************************

      if Temp_Nd2 /= null then
```

```
      AddToFront (SLC, Temp_Nd1);
      SLC.Next.NumRef := -1;
      if Temp_Nd2.NumRef /= -1 then
         Temp_Nd3 := MakeNode (Temp_Nd2.Addr);
         Temp_Nd3.NumRef := Temp_Nd2.NumRef;
         Insert_In_Order (SLC_Ref_List, Temp_Nd3);
      end if;
      Delete (SLC, Temp_Nd2);

   else

   -- ****************************************
   -- Since a miss has occurred in the SLC,
   -- search for the address in the CAM.
   -- ****************************************

      SLC_Miss := SLC_Miss + 1;
      CAM_Total_Refs := CAM_Total_Refs + 1;
      CAM_Size_Index := index (CAM_Size);

      if Temp_CAM_Size /= 0 then
         for i in 1 .. Temp_CAM_Size loop
            j := index (i);
            if CAM.Ref_Count (j) /= -1 then
               CAM.Ref_Count (j) := CAM.Ref_Count (j) + 1;
            end if;
         end loop;
      end if;

      SearchQ (CAM, n, CAM_Size_Index, Position, Found);

   -- ****************************************
   -- If the address is found in the CAM, then
   -- prefetch a line from the CAM into the SLC.
   -- The SLC line is comprised of the requested
   -- address + the addresses located after the
   -- requested address (total equal to the SLC
   -- line size).  This action represents a
   -- spatial prefetch of the structural
   -- locality captured in the CAM.
   -- ****************************************

      if Found = true then          -- address found in CAM

         if CAM.Ref_Count (Position) /= -1 then
            Temp_Nd1 := MakeNode (CAM.Address (Position));
            Temp_Nd1.NumRef := CAM.Ref_Count (Position);
            Insert_In_Order (CAM_Ref_List, Temp_Nd1);
            CAM.Ref_Count (Position) := -1;
         end if;
         while Position /= CAM.tail
           and then k /= SLC_Line_Size loop
               k := k + 1;
               Position := Position mod CAM_Size_Index + 1;
            end loop;

         for i in 1 .. k loop

         -- ****************************************
         -- If the SLC is full, then delete the
         -- address located at the rear of the list
         -- (LRU replacement).
         -- ****************************************

            if Temp_SLC_Size = SLC_Size then
```

```
            if SLC.tail.NumRef /= -1 then
               SLC_Non_Ref := SLC_Non_Ref + 1;
            end if;

            Delete (SLC, SLC.tail);
            Temp_SLC_Size := Temp_SLC_Size - 1;

         end if;

      -- ***************************************

         Temp_Ndl := MakeNode (CAM.Address (Position));
         AddToFront (SLC, Temp_Ndl);
         SLC_Total_Prefetch := SLC_Total_Prefetch + 1;
         Temp_SLC_Size := Temp_SLC_Size + 1;
         if Position = 1 then
            Position := CAM_Size_Index;
         else
            Position := Position - 1;
         end if;

      end loop;

   else                              -- address not found in CAM

   -- ***********************************************
   -- If a miss occurs in the CAM, then prefetch a
   -- line from main memory into the CAM.  The line
   -- will be comprised of the block of memory (equal
   -- to the CAM line size) in which the requested
   -- address is located.
   -- ***********************************************

      CAM_Miss := CAM_Miss + 1;

   -- ***********************************************
   -- If the CAM is full, then the delete the
   -- addresses (amount equal to the CAM line size)
   -- in the front of the CAM (FIFO replacement).
   -- ***********************************************

      if Temp_CAM_Size = CAM_Size then
         for i in 1 .. CAM_Line_Size loop

            if CAM.Ref_Count (CAM.head mod CAM_Size_Index + 1)
               /= -1 then
               CAM_Non_Ref := CAM_Non_Ref + 1;
            end if;

            Dequeue (CAM, CAM_Size_Index);
            Temp_CAM_Size := Temp_CAM_Size - 1;

         end loop;
      end if;

   -- ***********************************************

      o := n / CAM_Line_Size;
      CAM_Prefetch_Addr := o * CAM_Line_Size;

   -- ***********************************************
   -- CAM prefetch for a positive integer address.
   -- ***********************************************

      if n >= 0 then
         CAM_Prefetch_Addr := CAM_Prefetch_Addr - 1;
```

```
      for i in 1 .. CAM_Line_Size loop
         CAM_Prefetch_Addr := CAM_Prefetch_Addr + 1;
         Enqueue (CAM, CAM_Size_Index, CAM_Prefetch_Addr);
         CAM_Total_Prefetch := CAM_Total_Prefetch + 1;
         Temp_CAM_Size := Temp_CAM_Size + 1;
      end loop;
   else

   -- ********************************************
   -- CAM prefetch for a negative integer address.
   -- ********************************************

      if n rem CAM_Line_Size /= 0 then
         CAM_Prefetch_Addr := CAM_Prefetch_Addr - CAM_Line_Size;
      end if;
      for i in 1 .. CAM_Line_Size loop
         Enqueue (CAM, CAM_Size_Index, CAM_Prefetch_Addr);
         CAM_Total_Prefetch := CAM_Total_Prefetch + 1;
         CAM_Prefetch_Addr := CAM_Prefetch_Addr + 1;
         Temp_CAM_Size := Temp_CAM_Size + 1;
      end loop;
   end if;

-- ********************************************
-- Prefetch the addresses from the CAM to the SLC
-- starting with the requested address and ending
-- with the last (tail of the circular queue)
-- address in the CAM.
-- ********************************************

   Temp_CAM_Tail := CAM.tail;

   while CAM.Address (Temp_CAM_Tail) /= n loop
      Temp_Nd1 := MakeNode (CAM.Address (Temp_CAM_Tail));

      if Temp_SLC_Size = SLC_Size then

         if SLC.tail.NumRef /= -1 then
            SLC_Non_Ref := SLC_Non_Ref + 1;
         end if;

         Delete (SLC, SLC.tail);
         Temp_SLC_Size := Temp_SLC_Size - 1;

      end if;

      AddToFront (SLC, Temp_Nd1);
      SLC_Total_Prefetch := SLC_Total_Prefetch + 1;
      Temp_SLC_Size := Temp_SLC_Size + 1;
      if Temp_CAM_Tail = 1 then
         Temp_CAM_Tail := CAM_Size_Index;
      else
         Temp_CAM_Tail := Temp_CAM_Tail - 1;
      end if;

   end loop;

   if Temp_SLC_Size = SLC_Size then

      if SLC.tail.NumRef /= -1 then
         SLC_Non_Ref := SLC_Non_Ref + 1;
      end if;

      Delete (SLC, SLC.tail);
      Temp_SLC_Size := Temp_SLC_Size - 1;
```

```
            end if;

            Temp_Nd1 := MakeNode (CAM.Address (Temp_CAM_Tail));
            AddToFront (SLC, Temp_Nd1);
            SLC_Total_Prefetch := SLC_Total_Prefetch + 1;
            Temp_SLC_Size := Temp_SLC_Size + 1;

         end if;

      end if;

   end Serv_Data_Write;
end Serv_Data_Write_Package;
```

```
with Addr_Record_Package,
     LinkedLists_Package, CircularQ_Package, Text_IO;
use  Addr_Record_Package,
     LinkedLists_Package, CircularQ_Package, Text_IO;

package Serv_Instr_Fetch_Package is

  procedure Serv_Instr_Fetch
            (Input_File           : in File_Type;
             Memory_Ref           : in Addr_Record;
             SLC                  : in out List;
             CAM                  : in out Queue;
             SLC_Ref_List         : in out List;
             CAM_Ref_List         : in out List;
             SLC_Miss             : in out natural;
             CAM_Miss             : in out natural;
             SLC_Total_Refs       : in out natural;
             CAM_Total_Refs       : in out natural;
             SLC_Non_Ref          : in out natural;
             CAM_Non_Ref          : in out natural;
             SLC_Total_Prefetch   : in out natural;
             CAM_Total_Prefetch   : in out natural;
             SLC_Size             : in natural;
             CAM_Size             : in natural;
             SLC_Line_Size        : in natural;
             CAM_Line_Size        : in natural;
             Temp_SLC_Size        : in out natural;
             Temp_CAM_Size        : in out natural);

end Serv_Instr_Fetch_Package;
```

```
package body Serv_Instr_Fetch_Package is

   procedure Serv_Instr_Fetch
            (Input_File          : in File_Type;
             Memory_Ref          : in Addr_Record;
             SLC                 : in out List;
             CAM                 : in out Queue;
             SLC_Ref_List        : in out List;
             CAM_Ref_List        : in out List;
             SLC_Miss            : in out natural;
             CAM_Miss            : in out natural;
             SLC_Total_Refs      : in out natural;
             CAM_Total_Refs      : in out natural;
             SLC_Non_Ref         : in out natural;
             CAM_Non_Ref         : in out natural;
             SLC_Total_Prefetch  : in out natural;
             CAM_Total_Prefetch  : in out natural;
             SLC_Size            : in natural;
             CAM_Size            : in natural;
             SLC_Line_Size       : in natural;
             CAM_Line_Size       : in natural;
             Temp_SLC_Size       : in out natural;
             Temp_CAM_Size       : in out natural) is

      package Index_Integer_IO is new integer_IO (index);
      use Index_Integer_IO;

      Reference            : Addr_Record;
      Temp_Nd1, Temp_Nd2   : NodePointer;
      Temp_Nd3             : NodePointer;
      Position             : index := CAM.head;
      Found                : boolean := false;
      k                    : natural := 1;
      CAM_Prefetch_Addr    : integer;
      n,o,p                : integer;
      CAM_Size_Index, j    : index;
      Temp_CAM_Tail        : index;

   begin
   -- ****************************************
   -- Search for memory reference in the SLC.
   -- ****************************************

      SLC_Total_Refs := SLC_Total_Refs + 1;

      Temp_Nd1 := SLC.Next;
      while Temp_Nd1 /= null loop
         if Temp_Nd1.NumRef /= -1 then
            Temp_Nd1.NumRef := Temp_Nd1.NumRef + 1;
         end if;
         Temp_Nd1 := Temp_Nd1.Next;
      end loop;

      n := Memory_Ref.Address;
      Temp_Nd1 := MakeNode (Memory_Ref.Address);

      Temp_Nd2 := Search (SLC, n);

   -- ****************************************
   -- If the address is found in the SLC,
   -- then delete the address in the cache
   -- (linked list) and add it to the front
   -- of the list (most recently used).
   -- ****************************************

      if Temp_Nd2 /= null then
```

```
      AddToFront (SLC, Temp_Nd1);
      SLC.Next.NumRef := -1;
      if Temp_Nd2.NumRef /= -1 then
         Temp_Nd3 := MakeNode (Temp_Nd2.Addr);
         Temp_Nd3.NumRef := Temp_Nd2.NumRef;
         Insert_In_Order (SLC_Ref_List, Temp_Nd3);
      end if;
      Delete (SLC, Temp_Nd2);

   else

   -- ****************************************
   -- Since a miss has occurred in the SLC,
   -- search for the address in the CAM.
   -- ****************************************

      SLC_Miss := SLC_Miss + 1;
      CAM_Total_Refs := CAM_Total_Refs + 1;
      CAM_Size_Index := index (CAM_Size);

      if Temp_CAM_Size /= 0 then
         for i in 1 .. Temp_CAM_Size loop
            j := index (i);
            if CAM.Ref_Count (j) /= -1 then
               CAM.Ref_Count (j) := CAM.Ref_Count (j) + 1;
            end if;
         end loop;
      end if;

      SearchQ (CAM, n, CAM_Size_Index, Position, Found);

   -- ****************************************
   -- If the address is found in the CAM, then
   -- prefetch a line from the CAM into the SLC.
   -- The SLC line is comprised of the requested
   -- address + the addresses located after the
   -- requested address (total equal to the SLC
   -- line size).  This action represents a
   -- spatial prefetch of the structural
   -- locality captured in the CAM.
   -- ****************************************

      if Found = true then          -- address found in CAM

         if CAM.Ref_Count (Position) /= -1 then
            Temp_Nd1 := MakeNode (CAM.Address (Position));
            Temp_Nd1.NumRef := CAM.Ref_Count (Position);
            Insert_In_Order (CAM_Ref_List, Temp_Nd1);
            CAM.Ref_Count (Position) := -1;
         end if;
         while Position /= CAM.tail
            and then k /= SLC_Line_Size loop
               k := k + 1;
               Position := Position mod CAM_Size_Index + 1;
            end loop;

         for i in 1 .. k loop

         -- ****************************************
         -- If the SLC is full, then delete the
         -- address located at the rear of the list
         -- (LRU replacement).
         -- ****************************************

            if Temp_SLC_Size = SLC_Size then
```

```
            if SLC.tail.NumRef /= -1 then
               SLC_Non_Ref := SLC_Non_Ref + 1;
            end if;

            Delete (SLC, SLC.tail);
            Temp_SLC_Size := Temp_SLC_Size - 1;

         end if;

      -- ***************************************

         Temp_Ndl := MakeNode (CAM.Address (Position));
         AddToFront (SLC, Temp_Ndl);
         SLC_Total_Prefetch := SLC_Total_Prefetch + 1;
         Temp_SLC_Size := Temp_SLC_Size + 1;
         if Position = 1 then
            Position := CAM_Size_Index;
         else
            Position := Position - 1;
         end if;

      end loop;

   else                              -- address not found in CAM

   -- ********************************************
   -- If a miss occurs in the CAM, then prefetch a
   -- line from main memory into the CAM.  The line
   -- will be comprised of the block of memory (equal
   -- to the CAM line size) in which the requested
   -- address is located.
   -- ********************************************

      CAM_Miss := CAM_Miss + 1;

   -- ********************************************
   -- If the CAM is full, then the delete the
   -- addresses (amount equal to the CAM line size)
   -- in the front of the CAM (FIFO replacement).
   -- ********************************************

      if Temp_CAM_Size = CAM_Size then
         for i in 1 .. CAM_Line_Size loop

            if CAM.Ref_Count (CAM.head mod CAM_Size_Index + 1)
               /= -1 then
               CAM_Non_Ref := CAM_Non_Ref + 1;
            end if;

            Dequeue (CAM, CAM_Size_Index);
            Temp_CAM_Size := Temp_CAM_Size - 1;

         end loop;
      end if;

   -- ********************************************

      o := n / CAM_Line_Size;
      CAM_Prefetch_Addr := o * CAM_Line_Size;

   -- ********************************************
   -- CAM prefetch for a positive integer address.
   -- ********************************************

      if n >= 0 then
         CAM_Prefetch_Addr := CAM_Prefetch_Addr - 1;
```

```
      for i in 1 .. CAM_Line_Size loop
         CAM_Prefetch_Addr := CAM_Prefetch_Addr + 1;
         Enqueue (CAM, CAM_Size_Index, CAM_Prefetch_Addr);
         CAM_Total_Prefetch := CAM_Total_Prefetch + 1;
         Temp_CAM_Size := Temp_CAM_Size + 1;
      end loop;
   else

   -- ********************************************
   -- CAM prefetch for a negative integer address.
   -- ********************************************

      if n rem CAM_Line_Size /= 0 then
         CAM_Prefetch_Addr := CAM_Prefetch_Addr - CAM_Line_Size;
      end if;
      for i in 1 .. CAM_Line_Size loop
         Enqueue (CAM, CAM_Size_Index, CAM_Prefetch_Addr);
         CAM_Total_Prefetch := CAM_Total_Prefetch + 1;
         CAM_Prefetch_Addr := CAM_Prefetch_Addr + 1;
         Temp_CAM_Size := Temp_CAM_Size + 1;
      end loop;
   end if;

-- ********************************************
-- Prefetch the addresses from the CAM to the SLC
-- starting with the requested address and ending
-- with the last (tail of the circular queue)
-- address in the CAM.
-- ********************************************

   Temp_CAM_Tail := CAM.tail;

   while CAM.Address (Temp_CAM_Tail) /= n loop
      Temp_Ndl := MakeNode (CAM.Address (Temp_CAM_Tail));

      if Temp_SLC_Size = SLC_Size then

         if SLC.tail.NumRef /= -1 then
            SLC_Non_Ref := SLC_Non_Ref + 1;
         end if;

         Delete (SLC, SLC.tail);
         Temp_SLC_Size := Temp_SLC_Size - 1;

      end if;

      AddToFront (SLC, Temp_Ndl);
      SLC_Total_Prefetch := SLC_Total_Prefetch + 1;
      Temp_SLC_Size := Temp_SLC_Size + 1;
      if Temp_CAM_Tail = 1 then
         Temp_CAM_Tail := CAM_Size_Index;
      else
         Temp_CAM_Tail := Temp_CAM_Tail - 1;
      end if;

   end loop;

   if Temp_SLC_Size = SLC_Size then

      if SLC.tail.NumRef /= -1 then
         SLC_Non_Ref := SLC_Non_Ref + 1;
      end if;
```



```
      Delete (SLC, SLC.tail);
      Temp_SLC_Size := Temp_SLC_Size - 1;
```

```
            end if;

            Temp_Nd1 := MakeNode (CAM.Address (Temp_CAM_Tail));
            AddToFront (SLC, Temp_Nd1);
            SLC_Total_Prefetch := SLC_Total_Prefetch + 1;
            Temp_SLC_Size := Temp_SLC_Size + 1;

         end if;

      end if;

   end Serv_Instr_Fetch;
end Serv_Instr_Fetch_Package;
```

```
with Text_IO, Addr_Record_Package;
use  Text_IO, Addr_Record_Package;

package Fetch_Address_Package is

   procedure Load_Record (Input_File: in out File_Type;
                          Memory_Ref: out Addr_Record);

end Fetch_Address_Package;
```

```
with Text_IO, Hex_to_Dec_Package;
use  Text_IO, Hex_to_Dec_Package;

package body Fetch_Address_Package is

   procedure Load_Record (Input_File: in out File_Type;
                          Memory_Ref: out Addr_Record) is

      type Field_Type is (field1, field2);

      In_String      : string (1 .. 11);
      In_Length      : natural;
      New_Space      : natural;
      Last_Space     : natural;
      In_Field       : Field_Type := field1;
      Hex_Addr       : string (1 .. 8);
      Hex_Length     : natural;

   begin
         get_line (Input_File, In_String, In_Length);

         New_Space := 1;
         Last_Space := 0;

      loop

         while ((New_Space < In_Length) and
            (In_String (New_Space .. New_Space) /= " ")) loop
            New_Space := New_Space + 1;
         end loop;

         case In_Field is

            when field1 => Memory_Ref.The_Type :=
               In_String (New_Space - 1);
               In_Field := field2;

            when field2 => Hex_Addr ((Last_Space - 1) .. New_Space - 2) :=
               In_String ((Last_Space + 1) .. New_Space);
               Hex_Length := New_Space - Last_Space;
               Memory_Ref.Address := Hex_to_Dec (Hex_Addr, Hex_Length);
               In_Field := field1; exit;

            when others => exit;

         end case;

         Last_Space := New_Space;
         New_Space := New_Space + 1;

      end loop;
   end Load_Record;
end Fetch_Address_Package;
```

```
package  Hex_to_Dec_Package is

   function Hex_to_Dec (Hex_Addr: string; Hex_Length: natural)
                        return integer;

end Hex_to_Dec_Package;
```

```
with Text_IO; use Text_IO;

package body  Hex_to_Dec_Package  is

   function Hex_to_Dec (Hex_Addr: string; Hex_Length: natural)
                        return integer is

      Zero_Pos           : constant := character'pos ('0');
      Capital_A_Pos      : constant := character'pos ('A');
      Small_A_Pos        : constant := character'pos ('a');

      package Type_Integer_IO is new integer_IO (integer);
      use Type_Integer_IO;

      type Dec_Value_Type is range -2**31 .. 2**31-1;

      Temp_Dec_Value:  Dec_Value_Type := 0;
      Num_Value:  Dec_Value_Type range 0 .. 15;
      Dec_Value: integer;

      Hex_Char:  character;

   begin
      for i in 1 .. Hex_Length loop

         Hex_Char := Hex_Addr (i);

         case Hex_Char is

            when '0' .. '9' =>
               Num_Value := character'pos (Hex_Char) - Zero_Pos;

            when 'A' .. 'F' =>
               Num_Value := character'pos (Hex_Char) -
                            Capital_A_Pos + 10;

            when 'a' .. 'f' =>
               Num_Value := character'pos (Hex_Char) -
                            Small_A_Pos + 10;

            when others => exit;

         end case;

         if i < 8 then
            Temp_Dec_Value := 16 * Temp_Dec_Value + Num_Value;
         else
            Temp_Dec_Value := Temp_Dec_Value - 2**27;
            Temp_Dec_Value := 16 * Temp_Dec_Value + Num_Value;
         end if;

      end loop;

      Dec_Value := integer (Temp_Dec_Value);
      return Dec_Value;

   end Hex_to_Dec;

end Hex_to_Dec_Package;
```

```
with Addr_Record_Package; use Addr_Record_Package;

package Determine_Type_Package is

   function Address_Type (Memory_Ref: Addr_Record) return
character;

end Determine_Type_Package;
```

```
with Addr_Record_Package; use Addr_Record_Package;

package body Determine_Type_Package is

   function Address_Type (Memory_Ref: Addr_Record)
                          return character is
   begin
      return Memory_Ref.The_Type;
   end Address_Type;

end Determine_Type_Package;
```

```
with Text_IO; use Text_IO;

package Compute_Miss_Ratios_Package is

   procedure Compute_Miss_Ratios (SLC_Miss: in natural;
      CAM_Miss: in natural; SLC_Total_Refs: in natural;
      CAM_Total_Refs: in natural; Num_Ref: in natural;
      SLC_MR: out float; CAM_MR: out float;
      Output_File: in out File_Type);

end Compute_Miss_Ratios_Package;
```

```
with Text_IO;  use Text_IO;

package body Compute_Miss_Ratios_Package is

   procedure Compute_Miss_Ratios (SLC_Miss: in natural;
      CAM_Miss: in natural; SLC_Total_Refs: in natural;
      CAM_Total_Refs: in natural; Num_Ref : in natural;
      SLC_MR: out float; CAM_MR: out float;
      Output_File: in out File_Type) is

      type MR_Type is delta 0.0001 range 0.0 .. 1.0;

      package MR_Type_IO is new fixed_IO (MR_Type);
      package Type_Integer_IO is new integer_IO (integer);
      use MR_Type_IO, Type_Integer_IO;

      SLC_Miss_Ratio    : MR_Type;
      CAM_Miss_Ratio    : MR_Type;
      SM, CM, ST, CT    : float;
      SMR, CMR          : float;

   begin
      SM := float (SLC_Miss);
      CM := float (CAM_Miss);
      ST := float (SLC_Total_Refs);
      CT := float (CAM_Total_Refs);

      new_line (3);
      put ("Number of References Processed:  "); put (Num_Ref);
      new_line (2);
      put ("Number of SLC Misses:  "); put (SLC_Miss);
      new_line;
      put ("Number of CAM Misses:  "); put (CAM_Miss);
      new_line;
      put ("Total SLC References:  "); put (SLC_Total_Refs);
      new_line;
      put ("Total CAM References:  "); put (CAM_Total_Refs);
      new_line (2);

      SMR := SM / ST;
      CMR := CM / CT;

      SLC_Miss_Ratio := MR_Type (SMR);
      CAM_Miss_Ratio := MR_Type (CMR);
      SLC_MR := float (SLC_Miss_Ratio);
      CAM_MR := float (CAM_Miss_Ratio);

      put ("SLC_Miss_Ratio:  "); put (SLC_Miss_Ratio);
      new_line;
      put ("CAM_Miss_Ratio:  "); put (CAM_Miss_Ratio);

      Set_Col (Output_File, 1);
      put (Output_File, Num_Ref);
      put (Output_File, SLC_Miss);
      put (Output_File, CAM_Miss);
      put (Output_File, SLC_Total_Refs);
      put (Output_File, CAM_Total_Refs);
      put (Output_File, SLC_Miss_Ratio);
      put (Output_File, CAM_Miss_Ratio);

   end Compute_Miss_Ratios;

end Compute_Miss_Ratios_Package;
```

```
with Text_IO; use Text_IO;

package Compute_Memory_Access_Time_Package is

   procedure Compute_Memory_Access_Time (SLC_MR: in float;
                CAM_MR: in float; Output_File: in out File_Type);

end Compute_Memory_Access_Time_Package;
```

```
with Text_IO; use Text_IO;

package body Compute_Memory_Access_Time_Package is

   procedure Compute_Memory_Access_Time (SLC_MR: in float;
                CAM_MR: in float; Output_File: in out File_Type) is

      type Avg_Access_Type is delta 0.001 range 1.0 .. 32.0;

      package Type_Access_IO is new fixed_IO (Avg_Access_Type);
      use Type_Access_IO;

      Eff_Mem_Access       : Avg_Access_Type;
      CAM_Access           : float := 4.0;
      MM_Access            : float := 32.0;

   begin

      Eff_Mem_Access := Avg_Access_Type ((1.0 - SLC_MR) + (CAM_Access
         * (SLC_MR) * (1.0 - CAM_MR)) + (MM_Access * (SLC_MR) * (CAM_MR)));
      Set_Col (Output_File, 1);
      put (Output_File, "Effective Memory Access Time:  ");
      put (Output_File, Eff_Mem_Access);

   end Compute_Memory_Access_Time;

end Compute_Memory_Access_Time_Package;
```

```
with Text_IO, LinkedLists_Package, CircularQ_Package;
use  Text_IO, LinkedLists_Package, CircularQ_Package;

package Compute_Cache_Pollution_Package is

   procedure Compute_Cache_Pollution (SLC: in out List; CAM: in out Queue;
                                      SLC_Non_Ref: in out natural;
                                      CAM_Non_Ref: in out natural;
                                      SLC_Total_Prefetch: in natural;
                                      CAM_Total_Prefetch: in natural;
                                      Temp_CAM_Size: in natural;
                                      Output_File: in out File_Type);

end Compute_Cache_Pollution_Package;
```

```
package body Compute_Cache_Pollution_Package is

   procedure Compute_Cache_Pollution (SLC: in out List; CAM: in out Queue;
                                      SLC_Non_Ref: in out natural;
                                      CAM_Non_Ref: in out natural;
                                      SLC_Total_Prefetch: in natural;
                                      CAM_Total_Prefetch: in natural;
                                      Temp_CAM_Size: in natural;
                                      Output_File: in out File_Type) is

      type Fixed_Type is delta 0.0001 range 0.0 .. 1.0;

      package Type_Fixed_IO is new fixed_IO (Fixed_Type);
      use Type_Fixed_IO;
      package Index_Integer_IO is new integer_IO (index);
      use Index_Integer_IO;

      SLC_Pollution          : Fixed_Type;
      CAM_Pollution          : Fixed_Type;
      SNR, CNR, STP, CTP     : float;
      Temp_Node              : NodePointer;
      j                      : index;

      begin
         Temp_Node := SLC.Next;
         while Temp_Node /= null loop
            if Temp_Node.NumRef /= -1 then
               SLC_Non_Ref := SLC_Non_Ref + 1;
            end if;
            Temp_Node := Temp_Node.Next;
         end loop;

         SNR := float (SLC_Non_Ref);
         STP := float (SLC_Total_Prefetch);
         SLC_Pollution := Fixed_Type (SNR/STP);
         Set_Col (Output_File, 1);
         put (Output_File, "SLC Pollution:  ");
         put (Output_File, SLC_Pollution);

         ------------------------------------------------
         for i in 1 .. Temp_CAM_Size loop
            j := index (i);
            if CAM.Ref_Count (j) /= -1 then
               CAM_Non_Ref := CAM_Non_Ref + 1;
            end if;
         end loop;

         CNR := float (CAM_Non_Ref);
         CTP := float (CAM_Total_Prefetch);
         CAM_Pollution := Fixed_Type (CNR/CTP);
         Set_Col (Output_File, 1);
         put (Output_File, "CAM Pollution:  ");
         put (Output_File, CAM_Pollution);

      end Compute_Cache_Pollution;

   end Compute_Cache_Pollution_Package;
```

```
with Text_IO, LinkedLists_Package;
use  Text_IO, LinkedLists_Package;

package Generate_Ref_Frequency_List_Package is

   procedure Generate_Ref_Frequency_List (SLC_Ref_List: in out List;
                                          CAM_Ref_List: in out List;
                                          SLC_Non_Ref: in natural;
                                          CAM_Non_Ref: in natural;
                                          Reference_File: in out File_Type);

end Generate_Ref_Frequency_List_Package;
```

```
package body Generate_Ref_Frequency_List_Package is

   procedure Generate_Ref_Frequency_List (SLC_Ref_List: in out List;
                                          CAM_Ref_List: in out List;
                                          SLC_Non_Ref: in natural;
                                          CAM_Non_Ref: in natural;
                                          Reference_File: in out File_Type) is

      package Type_Integer_IO is new integer_IO (integer);
      use Type_Integer_IO;

      Temp_Nd1, Temp_Nd2      : NodePointer;
      Reference_Count         : natural := 0;

   begin
      Set_Col (Reference_File, 1);
      put (Reference_File, "SLC:  Frequency of References");
      Set_Col (Reference_File, 1);
      put (Reference_File, "          0");
      put (Reference_File, SLC_Non_Ref);
      Temp_Nd1 := SLC_Ref_List.Next;
      Temp_Nd2 := SLC_Ref_List.Next;
      while Temp_Nd1 /= null loop
         while Temp_Nd1 /= null and then
            Temp_Nd1.NumRef = Temp_Nd2.NumRef loop
            Temp_Nd1 := Temp_Nd1.Next;
            Reference_Count := Reference_Count + 1;
         end loop;
         Set_Col (Reference_File, 1);
         put (Reference_File, Temp_Nd2.NumRef);
         put (Reference_File, Reference_Count);
         Temp_Nd2 := Temp_Nd1;
         Reference_Count := 0;
      end loop;
      ------------------------------------------------------

      Set_Col (Reference_File, 1);
      put (Reference_File, "     ");
      Set_Col (Reference_File, 1);
      put (Reference_File, "CAM:  Frequency of References");
      Set_Col (Reference_File, 1);
      put (Reference_File, "          0");
      put (Reference_File, CAM_Non_Ref);
      Temp_Nd1 := CAM_Ref_List.Next;
      Temp_Nd2 := CAM_Ref_List.Next;
      while Temp_Nd1 /= null loop
         while Temp_Nd1 /= null and then
            Temp_Nd1.NumRef = Temp_Nd2.NumRef loop
            Temp_Nd1 := Temp_Nd1.Next;
            Reference_Count := Reference_Count + 1;
         end loop;
         Set_Col (Reference_File, 1);
         put (Reference_File, Temp_Nd2.NumRef);
         put (Reference_File, Reference_Count);
         Temp_Nd2 := Temp_Nd1;
         Reference_Count := 0;
      end loop;

   end Generate_Ref_Frequency_List;

end Generate_Ref_Frequency_List_Package;
```

# Appendix B: Test Trace Mapping to Testing Requirements

**Test Trace Mapping to Testing Requirements (Figure 3.16)**

Note: Testing requirements are identified at the first point in the test trace where they are tested. The following test trace was designed for the VAX traces: Version 1 driver. To test the Version 2 driver for the TI Explorer traces, the addresses were changed to integers so the testing requirements were tested at the same points in the test trace.

Cache Test Parameters (words)

| | | | |
|---|---|---|---|
| SLC Size: | 8 | CAM Size: | 16 |
| SLC Block: | 4 | CAM Block: | 2 |

```
0 1AF76945    <=  I.A.1, I.A.3, I.A.4, II.A.2, II.B.2,
                  II.D.1, II.D.2

0 1AF76946

0 1AF76944    <=  II.B.1, II.C.1

1 2378BC12    <=  II.D.4

1 2378BC13    <=  II.A.1

0 1AF76946

0 1AF76947

1 2378BC14

0 1AF76948

0 1AF76945    <=  II.C.2, II.C.3

2 189CDF03

2 189CDF02    <=  II.C.4

1 2458F3B2

1 2458F3B3

1 2458F3B4

1 2458F3B5

1 2458F3B6    <=  II.D.2

0 9654EF24
```

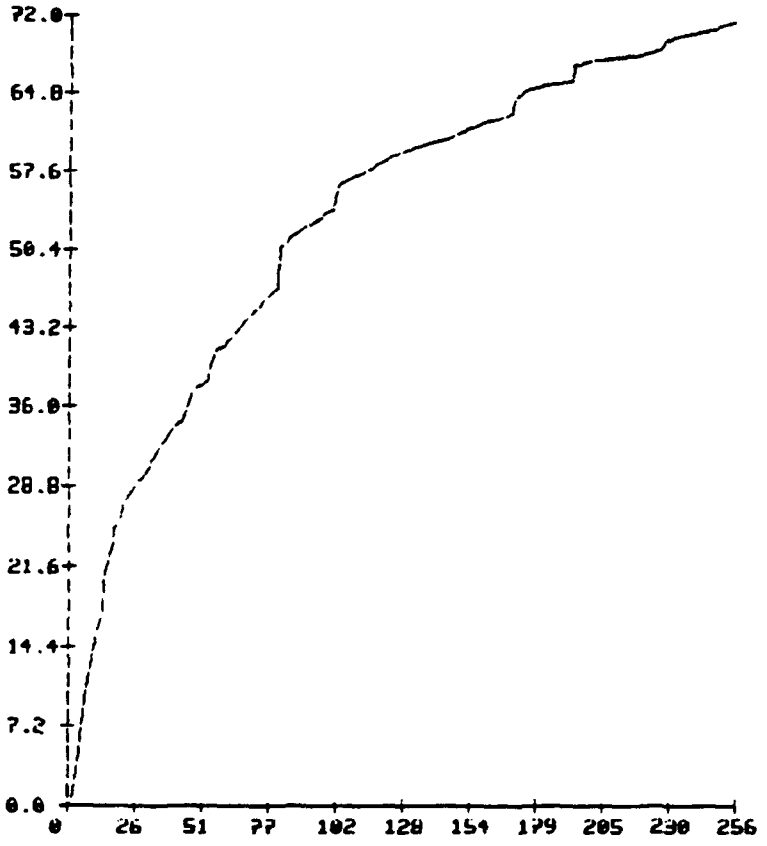
```
0 9654EF25
1 2458F3B2
1 2458F3B3
2 189CDF01
2 189CDF02
1 2458F3B4
1 2458F3B5     <=  II.E.1-4
3 08971234
4 367814BC
0 9654EF24
0 9654EF25
1 67209AF0
2 395CAFB1
0 9087BA23
1 65432DA0
1 2458F3B4
1 2458F3B5
0 56129027
1 9BF23467
1 67209AF0
2 395CAFB1
0 4F29B560
1 2A9F73CA
2 9876AFB3
0 68210BD3
0 56129027
```

```
1 9BF23467

2 B5DA0        <=  I.A.2

1 6754D231

0 4F29B560

1 2A9F73CA

0 1AF76945     <=  III.A-D
```

Test Trace Results

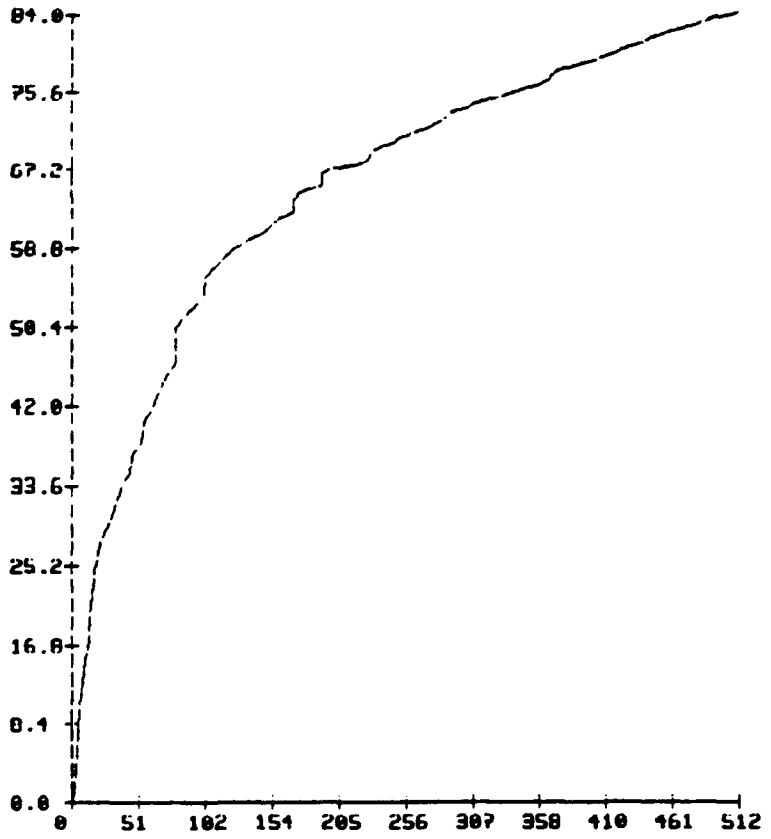
```
SLC Miss Rate:   0.7083
CAM Miss Rate:   0.7059
SLC Pollution:   0.8267
CAM Pollution:   0.8333
Eff Memory Access Time:   17.124

SLC Frequency of References:
 0        62       (never referenced)
 1         9
 3         1
 4         2
 7         1

CAM Frequency of References:
 0        40       (never referenced)
 1         1
 2         1
 4         1
 5         1
 6         2
 7         2
```

# Appendix C: Wc and Ws vs. Effective Cache Size Graphs




$W_S$

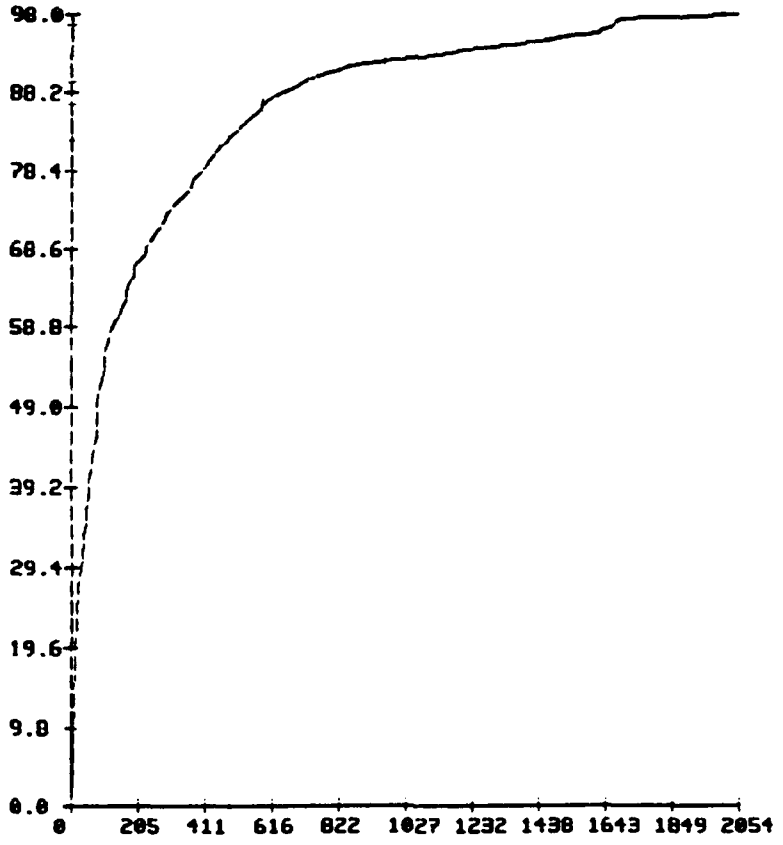
Effective Cache Size



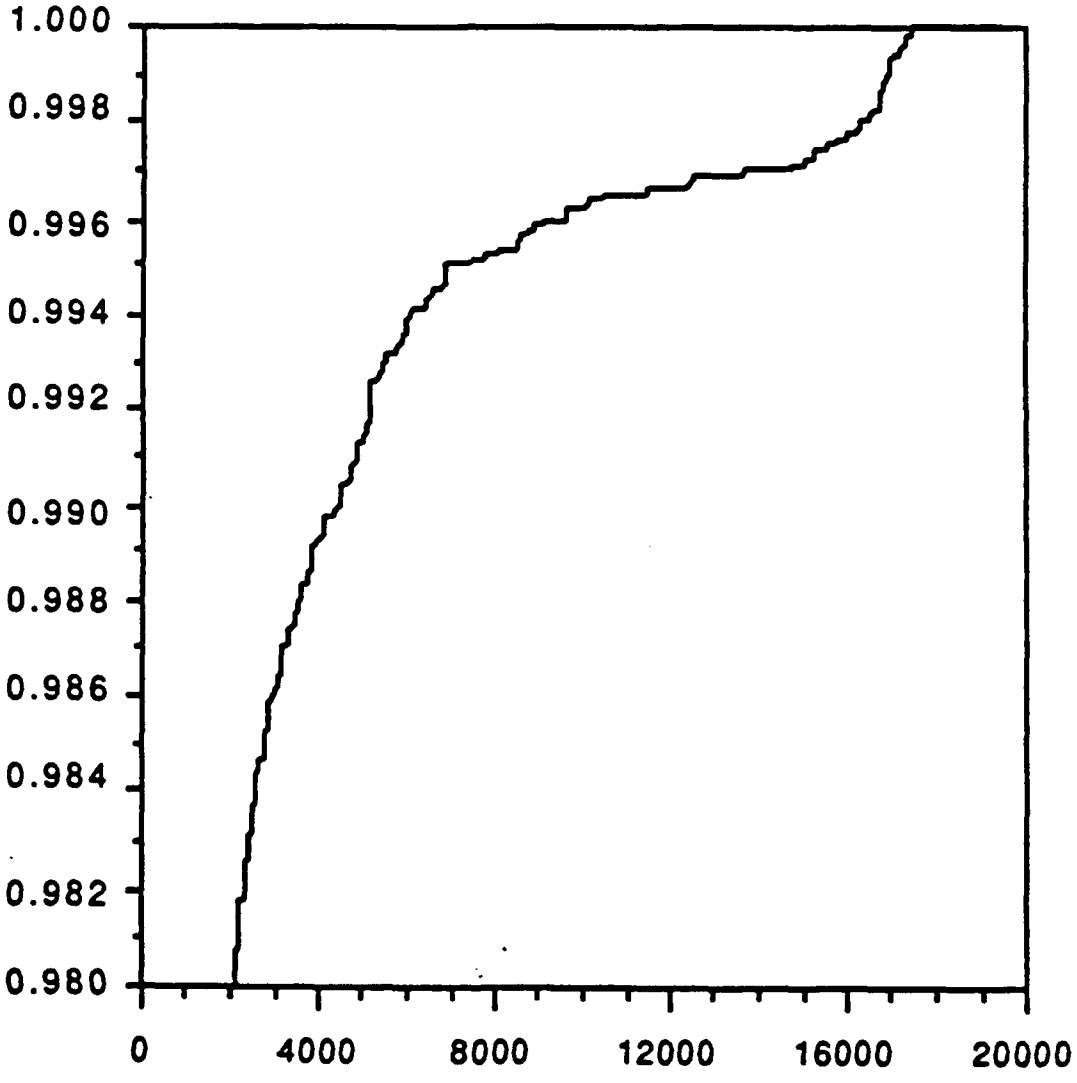

$W_S$

Effective Cache Size

$W_C$




Effective Cache Size


1.000
0.998
0.996
0.994
0.992
0.990
0.988
0.986
0.984
0.982
0.980
$W_C$
0
4000
8000
12000
16000
20000
Effective Cache Size

## Bibliography


Agarwal, A. and others. "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proceedings of the 13th Annual International Symposium on Computer Architecture*. 119-127. New York: IEEE Press, 1986.

Alexander, Cedell and others. "Cache Memory Performance in a Unix Environment," *ACM Computer Architecture News*, 14: 41-70 (June 1986).

Baer, Jean-Loup and others. "Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy," *Proceedings of the 16th Annual International Symposium on Computer Architecture*. 140-148. New York: IEEE Press, 1989.

Bakka, Bjorn O. and others. "Trace-Driven Simulations for a Two-Level Cache Design in Open Bus Systems," *ACM Computer Architecture News*, 18: 250-259 (June 1990).

Feldman, Michael B. *Data Structures with Ada*. Reston: Reston Publishing Company, Inc., 1985.

Hayes, John P. *Computer Architecture and Organization* (Second Edition). New York: McGraw-Hill Book Company, 1988.

Hennessy, John and others. "Performance Tradeoffs in Cache Design," *Proceedings of the 15th Annual International Symposium on Computer Architecture*. 290-298. New York: IEEE Press, 1988.

-----. "Characteristics of Performance-Optimal Multi-Level Cache Hierarchies," *Proceedings of the 16th Annual International Symposium on Computer Architecture*. 114-121. New York: IEEE Press, 1989.

Hill, Mark D. and Dionisios N. Pnevmatikatos. "Cache Performance of the Integer SPEC Benchmarks on a RISC," *ACM Computer Architecture News*, 18: 53-68 (June 1990).

Hobart, Maj William C., Jr. *An Investigation of the Locality of Memory Accesses During Symbolic Program Execution*. PhD dissertation. The University of Texas at Austin, Austin TX, 1989.

Iyer, Ravishankar K. and others. "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems," *IEEE Transactions on Computers*, 37: 1325-1335 (November 1988).

Johnson, Eric E. "Working Set Prefetching for Cache Memories," *ACM Computer Architecture News*, 17: 137-141 (December 1989).

Levy, Henry M. and Robert T. Short. "A Simulation Study of Two-Level Caches," *ACM Computer Architecture News*, 16: 81-87 (May 1988).

Przybylski, Steven. "The Performance Impact of Block Sizes and Fetch Strategies," *Proceedings of the 17th Annual International Symposium on Computer Architecture*. 160-169. New York: IEEE Press, 1990.

Smith, Alan J. "Sequentiality and Prefetching in Database Systems," *ACM Transactions on Database Systems*," 3: 223-247 (September 1978).

-----. "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computers*, 12: 7-21 (December 1978).

-----. "Cache Memories," *Computing Surveys*, 14: 473-523 (September 1982).

-----. "Line (Block) Size Choice for CPU Cache Memories," *IEEE Transactions on Computers*, C-36: 1063-1075 (September 1987).

Thazhuthaveetil, M.J. *A Structured Memory Access Architecture for Lisp. PhD dissertation*. The University of Wisconsin-Madison, Madison WI, 1986.